ProFoUnd: Program-analysis–based Form Understanding

Michael Benedikt, Tim Furche, Andreas Savvides
Oxford University
Oxford, United Kingdom
firstname.lastname@cs.ox.ac.uk

Pierre Senellart
Institut Télécom; Télécom ParisTech; CNRS LTCI
Paris, France
pierre.senellart@telecom-paristech.fr

ABSTRACT

An important feature of web search interfaces are the restrictions enforced on input values – those reflecting either the semantics of the data or requirements specific to the interface. Both integrity constraints and “access restrictions” can be of great use to web exploration tools. We demonstrate here a novel technique for discovering constraints that requires no form submissions whatsoever. We work via statically analyzing the JavaScript client-side code used to enforce the constraints, when such code is available. We combine custom recognizers for JavaScript functions relevant to constraint checking with a generic program analysis layer. Integrated with a web browser, our system shows the constraints detected on accessed web forms, and allows a user to see the corresponding JavaScript code fragment.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*JavaScript*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*

General Terms

Design,Experimentation

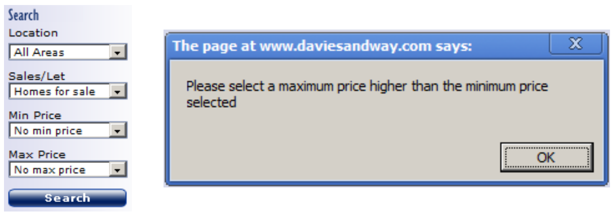
Keywords

deep Web, JavaScript, static analysis, Web form

1. INTRODUCTION

The *hidden web* or *deep web* refers to information accessible only via web forms. Search engines currently have only ad-hoc means of accessing this data, which is estimated to be several orders of magnitude larger than the surface web [2, 4]. Harvesting data from deep web interfaces has thus been an object of considerable study in the database and web communities. Much of the work has been on discovering the properties of web interfaces – determining the labels of form fields [7], identifying entry points to hidden web sources [1], identifying the domains of attributes [5], clustering form-based sites [8], or matching them to a known logical schema [3].

WWW 2012 Companion, April 16–20, 2012, Lyon, France.
ACM 978-1-4503-1230-1/12/04.



Please select a maximum price higher than the minimum price selected

Figure 1: Search form and JavaScript alert box on submission for `http://www.sharpeproperties.co.uk/`

An important feature of web search interfaces are the restrictions that they enforce on input values. These include integrity constraints that reflect restrictions on the underlying data – e.g., a UK postal code must be six characters, a salary should be a number, etc. They also include restrictions that reflect requirements that are specific to the interface. For example, a web-based telephone directory may require the user to enter a person’s name and street, returning all the matching phone numbers. Both integrity constraints and “access restrictions” can be of great use to web exploration tools. At the most obvious level, they restrict the way web search tools or aggregators can explore the web form; a crawler that is unaware of an access restriction or integrity constraint may make hundreds or thousands of unnecessary queries to the search engine. Indeed, without knowing integrity constraints, a crawler may never get to relevant data, since the number of queries necessary to find correct data by “brute force” will be prohibitive. Understanding constraints can also be useful in other form understanding tasks such as schema matching.

Despite the utility of integrity constraints and access restrictions, there has been little work on how to discover them. In this demonstration, we consider the common case where client-side code, written in JavaScript, enforces integrity constraints and access restrictions in the browser, before the submission of the form. Our system, ProFoUnd, (**Pro**gram-analysis–based **Fo**rm **Und**erstanding) discovers constraints by statically analyzing the JavaScript code, recognizes references to form fields, and detects comparisons that represent constraints.

Example 1. Consider the web form on the left in Figure 1. It enforces several constraints on the input, generating dialog boxes with error messages when these constraints are not satisfied, such as that shown on the right side of the figure. A deep web crawler or information extraction tool that navigated this page might perform many accesses to the site to obtain any valid entry. A tool that wanted to extract

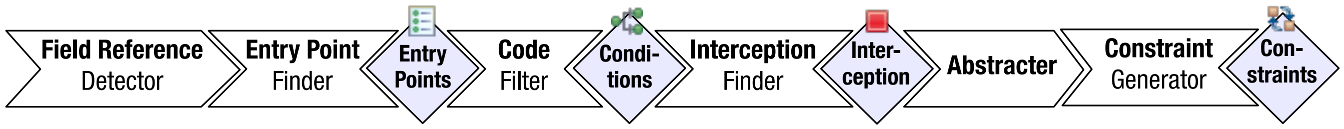


Figure 2: ProFoUnd's architecture

```
function CheckSimpleSearch(theForm) {
  if (theForm.minprice && theForm.maxprice) {
    if (!isNaN(theForm.minprice.value) &&
        !isNaN(theForm.maxprice.value)) {
      if (parseInt(theForm.minprice.value) >=
          parseInt(theForm.maxprice.value)) {
        alert('Please select a maximum price higher than the
            minimum price selected');
        return false;
  }}}
  return true;
}
```

Figure 3: JavaScript validation code

all valid data for the site would perform an enormous number of additional queries if it were unaware of the constraints.

The constraints involved here are all performed on the client side.[1] An excerpt of the code is in Figure 3. ProFoUnd is able to reverse-engineer the min < max constraint from analyzing the script, without needing any example. □

Analysis of scripting languages like JavaScript is known to be extremely difficult. JavaScript is weakly-typed and supports run-time modification of object structure. It allows the conversion of free-text to code (*eval*), a feature that is particularly problematic from the point of view of static analysis [6]. In addition, the client code that is visible, although perhaps not large, will often make use of huge JavaScript libraries. In the example of Figure 1, access functions are used to navigate to form elements, and a JavaScript validation function is attached to respond to some event (e.g., *click*, *submit*). These access functions are often come from popular cross-domain libraries (JavaScript *frameworks*), such as jQuery. Similarly the functions that raise the errors (e.g., *alert*) are not JavaScript primitives, but rather come either from the browser environment or from libraries.

We deal with the challenge by making use of a two-tiered approach. We have a set of custom matchers recognizing library commands performing basic operations critical to constraint-handling; these include error-handling routines and idioms for accessing form elements. This eliminates the necessity of analyzing code within standard libraries. We also apply routines for several basic program analyses, including control-flow and alias analysis. We put these together to get a solution to the constraint-detection problem.

In this demo we will present the first approach to extracting general client-side validation rules, We briefly introduce our extensible approach, and an embodiment that deals with the features of the most popular libraries and frameworks. To highlight that this is indeed usable for extracting client-side constraints, we briefly present some experimental results in Section 3. We then introduce in Section 4 our demonstration scenario, which involves interfacing ProFoUnd with a browser to show a user constraints identified in real web forms and the corresponding JavaScript code fragments.

[1]In a typical Web application, for security reasons, the constraints would also be enforced on the server side.

2. SYSTEM DESCRIPTION

ProFoUnd takes as input the URL of a web form, and outputs a set of assertions involving the form fields – in our current implementation, assertions are boolean combinations of atomic comparisons between form fields and constants.

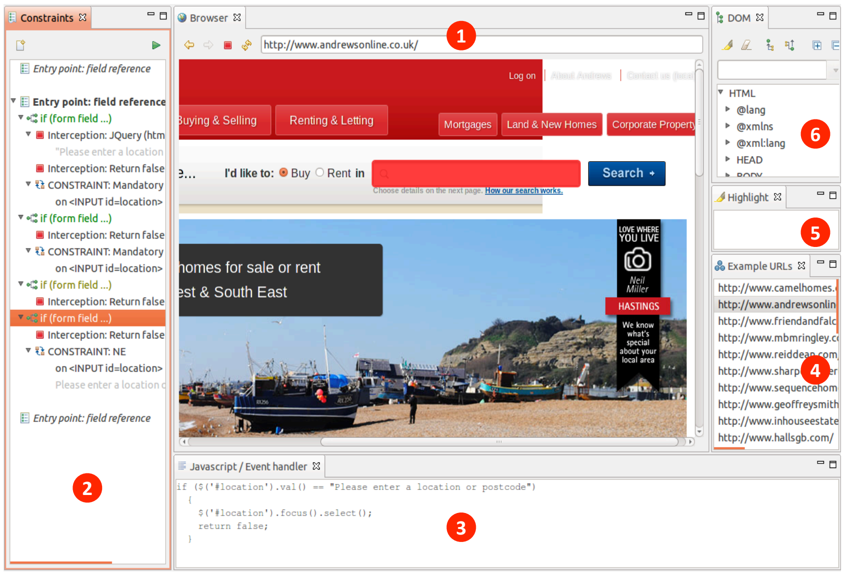
ProFoUnd is written in Java. Basic form analysis – e.g., detection of fields and their labels are provided by components from the DIADEM project.[2] DIADEM also provides APIs for accessing the live DOM of the form and manipulating form fields. JavaScript parsing is not provided by DIADEM, and thus ProFoUnd relies on the open-source Rhino parser.[3] An abstract syntax tree representing relevant code is created, which is the main artifact manipulated by the system.

No non-trivial JavaScript analysis problem can take into account JavaScript code in its entirety – it cannot deal with analyzing third-party libraries, and it cannot determine the semantics of arbitrary application code. Our approach thus proceeds by progressively simplifying the code structure, until we reach a language that is amenable to a traditional program analysis. In each simplification an approximation is performed, which can impact soundness or completeness – the guarantees we can give are thus only experimental. Our approach to third-party code that is re-used over many applications follows the two-tiered approach alluded to in the introduction. We classify the functionality of such libraries that is relevant to constraint validation, and on a per-library basis write "translators" that map library code into our global framework – the main functions we consider are identifying form fields, attaching code to form submission events, and raising errors. Our current system supports the libraries jQuery, YUI 2 & 3, Dojo, MooTools, and Prototype. Our approach to application-specific code is based on abstraction – we eliminate code of no interest for constraint validation, and we simplify features of code not amenable to exact analysis.

The resulting framework consists of several components. The diagram in Figure 2 shows how these components fit in together in order to extract constraints.

The most basic component is responsible for detecting *form field references* in the code and matching them with the corresponding form element. We identify for instance such code fragments as `document.getElementById("price")` in standard JavaScript, or `$("#price")` in jQuery, where *price* is the identifier of an `<input>` element in the HTML page. The next module is an *entry point finder*, whose function is to determine which top-level calls are associated with a form-submission event. This can be an `onsubmit` attribute on a `<form>` element, a *click* event dynamically bound to some button, etc. The *code filter* extracts the portion of the code that is relevant to form submission. If JavaScript code is directly embedded in an HTML form element's attribute, then the code filter just extracts that code. Otherwise the component traces forward from the entry points, pulling in code that needs to be analyzed. Within this code, blocks that are

[2]http://diadem-project.info/
[3]http://www.mozilla.org/rhino/

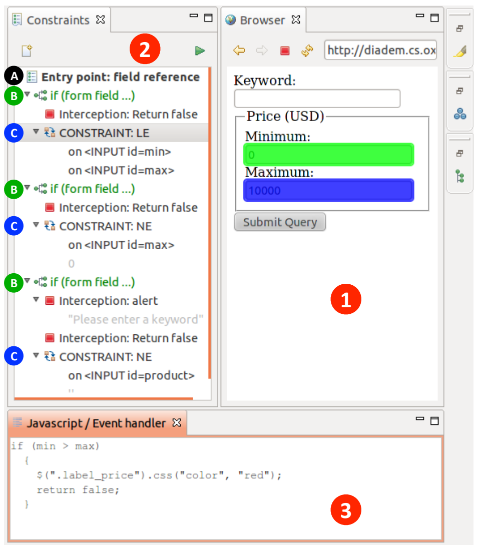


```
Entry point: field reference

▼ Entry point: field reference
  ▼ if (form field ...)
    ▼ Interception: JQuery (htm
        "Please enter a location
      Interception: Return false
    ▼ CONSTRAINT: Mandatory
        on <INPUT id=location>
  ▼ if (form field ...)
      Interception: Return false
    ▼ CONSTRAINT: Mandatory
        on <INPUT id=location>
  ▼ if (form field ...)
      Interception: Return false
  ▼ if (form field ...)
      Interception: Return false
    ▼ CONSTRAINT: NE
        on <INPUT id=location>
        Please enter a location c

Entry point: field reference
```

```
http://www.camelhomes.
http://www.andrewsonlin
http://www.friendandfalc
http://www.mbmringley.c
http://www.reiddean.com
http://www.sharp er
http://www.sequencehom
http://www.geoffreysmith
http://www.inhouseestate
http://www.hallsgb.com/
```

```
if ($('#location').val() == "Please enter a location or postcode")
  {
    $('#location').focus().select();
    return false;
  }
```

Figure 4: ProFoUnd interface



```
A Entry point: field reference
B ▼ if (form field ...)
        Interception: Return false
C     ▼ CONSTRAINT: LE
            on <INPUT id=min>
            on <INPUT id=max>
B ▼ if (form field ...)
        Interception: Return false
C     ▼ CONSTRAINT: NE
            on <INPUT id=max>
            0
B ▼ if (form field ...)
      ▼ Interception: alert
            "Please enter a keyword"
        Interception: Return false
C     ▼ CONSTRAINT: NE
            on <INPUT id=product>
            ''
```

```
if (min > max)
  {
    $(".label_price").css("color", "red");
    return false;
  }
```

Figure 5: ProFoUnd constraint and code views

relevant to constraint analysis are identified; form field references are one important feature, and these have already been detected by the field reference detector. The *interception finder* detects code blocks responsible for intercepting form submissions – informally, *error leg* code. Our interception finder makes use of recognizers for common idioms of blocking form submission (created as hand-coded patterns) along with a keyword-based analyzer that detects alert messages that indicate a constraint (e.g. "Please enter…").

With the important blocks identified, the decorated code is then abstracted to a simpler language, eliminating features that cannot be analyzed directly – this is the job of the *abstracter*. Finally, the resulting abstracted code is analyzed to determine which form references lead to constraints – this is the job of the *constraint generator*.

3. EVALUATION

ProFoUnd was tested on a number of randomly selected real-estate websites with deep web search interfaces.

A total of 70 real estate websites with deep web search interfaces were randomly selected from the repository created by the DIADEM project. For those 70 search interfaces, prior to the experiments, we manually identified that 30 of them had validation code on the client-side, while the other 40 had no client-side validation code. For the 30 search interfaces with client-side validation code, we manually identified all individual integrity constraints enforced on the client-side.

For the 40 websites with search interfaces that had no client-side validation code, we tested to see if ProFoUnd would return any constraints – i.e., any false positives. For the remaining 30 websites with client-side validation code, we tested both the precision and recall of ProFoUnd.

The result of the first experiment was that no false positives were returned. The key thing is that ProFoUnd correctly detected no interception events occur in any of these sites. For the 30 search interfaces with client-side validation code, ProFoUnd detected 22 of the 35 possible constraints correctly.

From running ProFoUnd, we were able to identify various different types of constraints, including:

- Mandatory fields enforced using standard JavaScript, others using the ASP.NET web framework, and still others using jQuery. Note that form interception and entry points use different idioms in each of these cases.
- Inter-field dependencies, such as the minimum and maximum price dependency shown in Figure 3. Note that this constraint makes use of both non-standard JavaScript (for accessing form fields) and calls to functions such as `parseInt()` and `isNaN()`.

The constraints that were missed were due to a mixture of factors: use of complex JavaScript features, such as *eval*; obfuscation of validation code, for example with tools such as the JavaScript Obfuscator[4] or JScrambler[5]. Note that obfuscation through variable renaming is not a problem for ProFoUnd since the analysis does not rely on variable names; obfuscation by introducing extra layers of computation (useless function calls, encryption of character strings, etc.) is. On a larger sample, we would expect that we would also miss constraints due to the limitations of our abstracter and constraint generator – the program analysis that these are based on is currently extremely naïve, employing no JavaScript-specific analysis techniques whatsoever. While we do not hope to be able to handle all occurrences of *eval*, in ongoing research we are investigating idiomatic uses of dynamic JavaScript features that can be incorporated into the ProFoUnd framework.

4. DEMONSTRATION DETAILS

ProFoUnd is accompanied by a visual GUI for inspecting the discovered constraints, as well as ProFoUnd's reasoning to discover these constraints. It is implemented as an Eclipse application with an embedded browser that allows the user

[4]`http://www.javascriptobfuscator.com/`
[5]`http://jscrambler.com/`

to browse to an arbitrary web page, execute ProFoUnd, and drill down to relevant fields and fragments of the JavaScript code where "relevance" follows the architecture from Figure 2.

Figure 4 illustrates the interface of the ProFoUnd GUI. It consists of six views: (1) The *browser view* contains a full-fledged Mozilla-based browser component. It is modified to allow highlighting of relevant form fields and of hovered elements, to provide easy insight into the HTML structure. (2) The *constraint view* shows all entry points, conditions, interception points, and constraints identified by ProFoUnd in a hierarchy. Entry points are code fragments or event handler attributes referencing a form field. For each entry point, we list all conditionals in the scope of that fragment or event handler, if they contain a reference to a form field. For each conditional, we list first the interception points such as error messages or `return false` and second the constraints identified by ProFoUnd. (3) The *code view* presents relevant HTML and JavaScript fragments for the selection in the constraint view. In Figure 4, a condition is selected and the corresponding JavaScript code is shown in (3).

The remaining views provide a list of URLs (4), control over highlighted elements (5), and detailed access to the DOM for drilling down to specific script elements or event handler attributes (6). Here, we focus on the first three.

For the demonstration, we have prepared a set of "interesting" examples in (4), but also allow the user to pick websites of his or her choosing. We will step through sites that exhibit the complex reasoning carried out by the system. Figure 4 shows the ProFoUnd GUI on a real-world UK real estate website with a fairly complex jQuery based JavaScript validator. This site has a constraint requiring that the single *location* input field is not empty. It also uses that field to display an error, by setting the value to *"Please enter a location or postcode"*, and in consequence also rejects any submission that has this "default" value. Thus there are two constraints: that location is not empty and that location is not equal to the error message. ProFoUnd successfully identifies these constraints: It finds three possible entry points, i.e., references to form fields, but recognizes correctly that only one of them (highlighted in bold) contains conditions and thus potentially constraints. The others are event handlers for auto-completion. Four conditions are identified, but only three (presented in green) have both interception points and constraints. The first two yield the same constraint (location not empty). In the JavaScript code they handle two different states of the validation: the first one provides the error message, the second merely sets the focus on the field. The final condition handles the case where the value of location is the error message. Below we show the first condition in the context of the entry point. The entry point binds an event handler to the *submit* event of the search button. The event handler checks if location is not empty (and the UI has a certain "stage", which is not reported as a constraint as it is not about a form field). The value of location is set to the error message and a click event handler is provided to remove the error message if the user clicks into the field. Finally, the form submission is prevented.

```
$('#search').bind('submit', function(e) {
 if ($('#location').val() == "" &&
      $('#searchRadiusTab').hasClass('tab current')) {
    $('#location').val("Please enter a location or
        postcode");
    $('#location').bind('click', function() {
      $(this).val("");
      $('#location').unbind('click').unbind('keypress'); });
    ...
    return false; }
  ... });
```

Figure 5 illustrates multi-field constraints, focusing on the constraint (2) and code (3) views. To keep the discussion and screenshot brief we choose an artificial form example. The form has a mandatory *keyword* field and *minimum* and *maximum price* fields where the minimum should be less than the maximum and the maximum should not be zero. This is a typical pattern for search forms in many product domains. As in the previous example, ProFoUnd correctly identifies all three constraints (C), each associated with a different condition (B) under the same entry point (A, the submit event handler on the form). The first two conditions use visual clues to communicate an error to the user, only the last one alerts the user with an error message, successfully recognized by ProFoUnd.

In the demonstration, we will also show examples for other typical form patterns, e.g., where event handlers are attached through attributes, with frameworks other than jQuery, or by complex auto-generated code from web frameworks such as ASP. A screencast of the demo can be found at `http://diadem.cs.ox.ac.uk/profound`.

5. ACKNOWLEDGMENTS

Benedikt is supported in part by EP/G004021/1 and EP/H017690/1 of the Engineering and Physical Sciences Research Council UK and in part by EC FP7-ICT-233599 (*FoX*). Furche is supported by ERC grant agreement 246858 *DIADEM*, while Senellart is supported by ERC grant agreement 226513 *Webdam*.

6. REFERENCES

[1] L. Barbosa and J. Freire. An adaptive crawler for locating hidden web entry points. In *WWW*, 2007.

[2] M. K. Bergman. The deep web: Surfacing hidden value. *J. Electronic Publishing*, 7, 2001.

[3] B. He, K. C.-C. Chang, and J. Han. Discovering complex matchings across web query interfaces: a correlation mining approach. In *KDD*, 2004.

[4] B. He, M. Patel, Z. Zhang, and K. C.-C. Chang. Accessing the deep web: A survey. *CACM*, 50(2):94–101, 2007.

[5] X. Jin, N. Zhang, and G. Das. Attribute domain discovery for hidden web databases. In *SIGMOD*, 2011.

[6] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do – a large-scale study of the use of eval in JavaScript applications. In *ECOOP*, 2011.

[7] J. Wang and F. H. Lochovsky. Data extraction and label assignment for web databases. In *WWW*, 2003.

[8] W. Wu, C. T. Yu, A. Doan, and W. Meng. An interactive clustering-based approach to integrating source query interfaces on the deep web. In *SIGMOD*, 2004.