ProFoUnd: Program-analysis-based Form Understanding

Authors

Michael Benedikt, Tim Furche, Andreas Savvides, Pierre Senellart

Digital Home

http://diadem.cs.ox.ac.uk/profound/

Contact

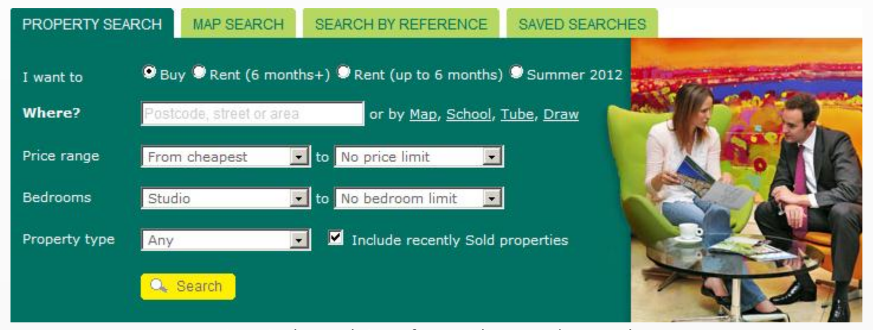
michael.benedikt@cs.ox.ac.uk
tim.furche@cs.ox.ac.uk
andreas.savvides@uk.ibm.com
pierre.senellart@telecom-paristech.fr

Motivation and Contributions

Background and Motivation

In the context of information extraction, in order to return all information that is relevant from a given website there are usually two necessary steps:

- Crawling the website, which is the traditional and more straightforward approach
- Tapping into the "deep" or "hidden" web via the available search interfaces



A typical search interface with some deep web content "hidden" behind it.

Dealing with a deep web search interface entails, roughly, the following steps:

- Find relevant website for a particular domain
- Identify a search interface (potentially leading to deep web content)
- Deduce input fields for an identified search interface and figure out corresponding meta-data with regards to labels and candidate domains of each such field
- Fill in the search form with appropriate input values and submit it (i.e. query a hidden database)

Querying a hidden database

Finding constraints associated with a particular search form prior to querying its hidden database (i.e. before submission) would be useful in a variety of different areas:

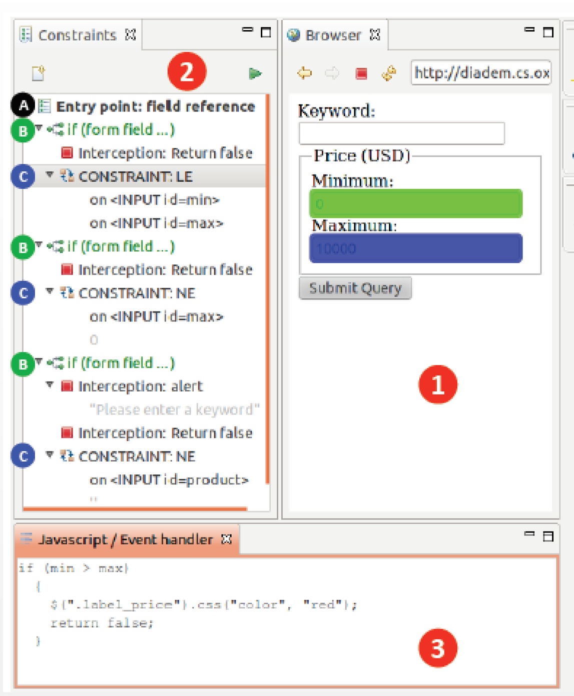
- Vertical search and information extraction
- Content surfacing
- Assistive technologies

This lead us to the following problem statement:

Given a search interface (i.e., a web form), can we infer integrity constraints from carrying out static JavaScript analysis?

Contributions

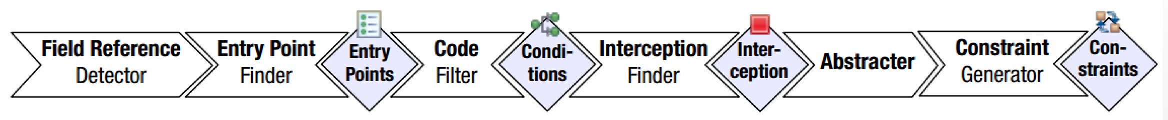
- A *novel* integrity constraint identification system based on *pattern matching* and *JavaScript analysis*.
- A system able to deal with a plethora of methods available for enforcing integrity constraints on the client-side (i.e., standard and non-standard JavaScript, JavaScript libraries and web frameworks)
- A system capable of deducing *relations* (conjunctive or disjunctive) amongst integrity constraints identified.
- *Thorough* experimental evaluation of ProFoUnd on a real-world data set, achieving integrity constraint identification with 100% accuracy and an F_1 score of over 0.77



ProFoUnd is a standalone system built in the context of: DIADEM

Architecture

ProFoUnd Architecture



We take a two-tiered approach of library-specific ad-hoc detectors coupled with program analysis.

Entry Points

Assignments of JavaScript function handlers to a particular event for a specific DOM element. The two most common ways to do such assignments:

- Attribute-based assignment – directly attached to the HTML:

```
<form name="propertysearch" action="/search.php" onsubmit="javascript:
    redirectPropertySearch();" method="post" id="propertysearch">
..
</form>
```

- Script-based assignment – programmatically doing the assignment with the help of a JavaScript library:

```
$('#propertysearch').bind('submit', function (event) {
  if($('#location').val() == "") {
    $('#locInfo').html("Please enter a location.");
    event.preventDefault();
  }
})
```

Conditions

Numerous syntactic statements can offer conditional behaviour; we focus on three JavaScript-specific:

- if – else
- Ternary operators
- Return statements involving comparison operators

Variables used in conditions might be aliased, hence aliasing analysis is used prior to searching for conditional statements

Interceptions

Clues which signal that a form may be restrained from being submitted, depending if a (set of) condition(s) is/are met. Two types of interceptions have been identified:

- Submission Interception – code that prevents the form from being submitted

```
event.preventDefault();
```

- Interception notification – code that notifies a user of the interception

```
$('#locInfo').html("Please enter a location.");
```

Error message analysis is used in order to confirm the analysis done up until this point or to identify a constraint.

From our annotated **Abstract Syntax Tree (AST)** we create a **Condition Control Flow Graph (CCFG)** in order to deduce what behaviour is yielded according to if a condition was met or not.

Constraints

Having identified client-side validation code based on the analysis, we need to translate conditional statements to corresponding constraints.

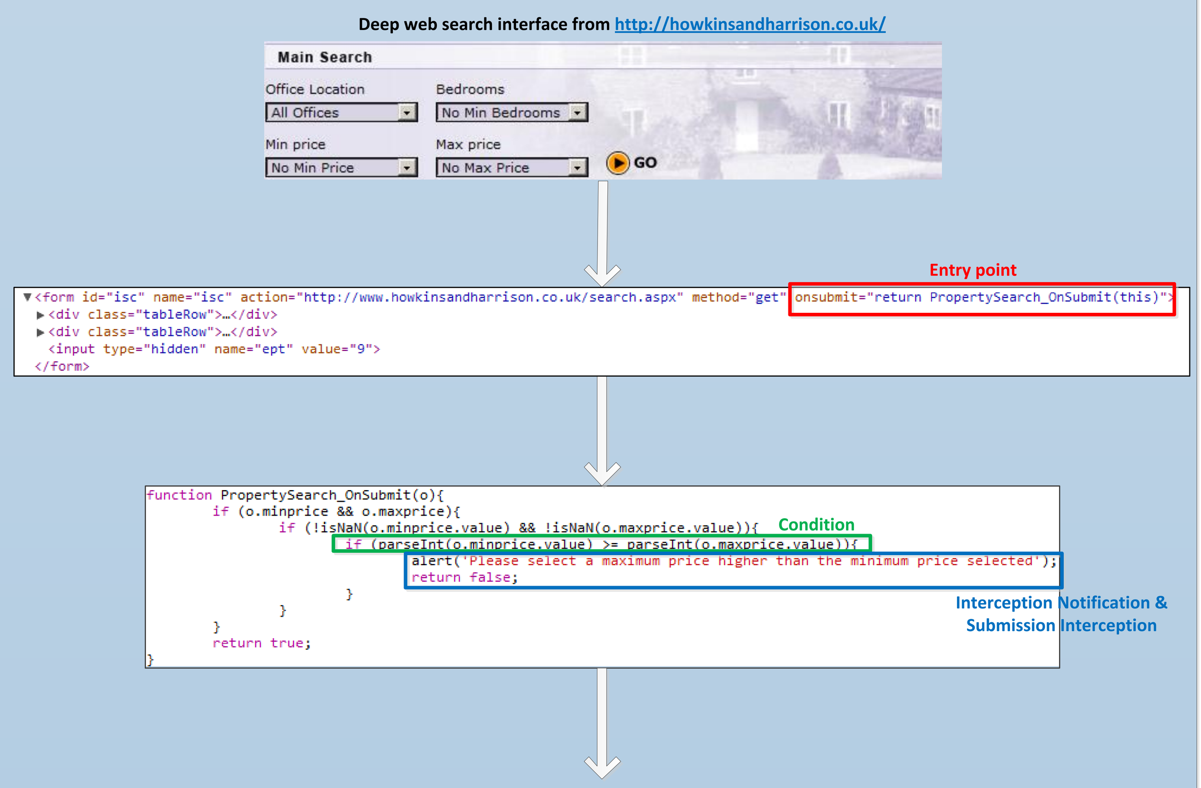
Binary constraints are first identified, followed by a refinement stage to deduce disjunctive constraints. The end result is a disjunction of constraints C_i as follows:

$$C_0 \lor C_1 \lor \ldots \lor C_{i-1} \lor C_i$$

Where each constraint C_i can be either:

- An atomic constraint A_i
- A conjunction of atomic constraints $A_0 \land A_1 \land \ldots \land A_{i-1} \land A_i$

An Example



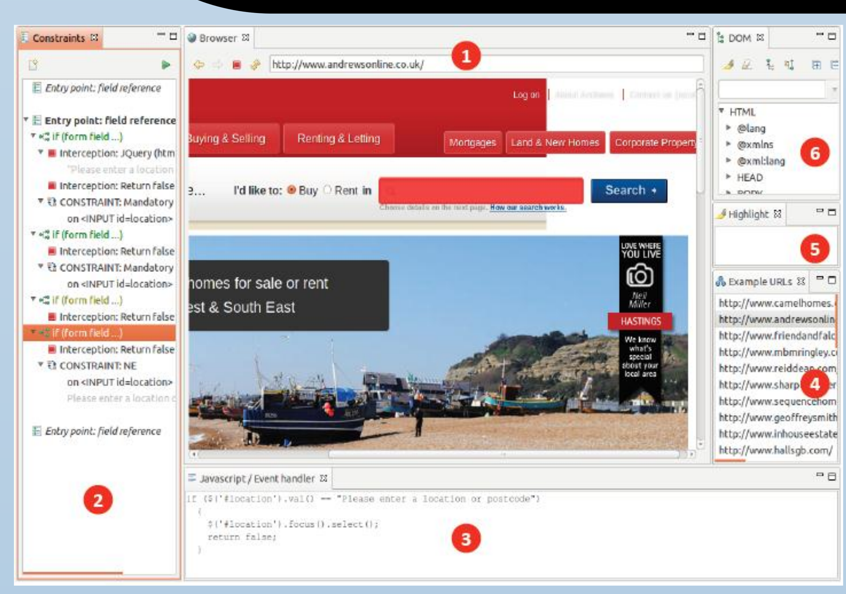
Constraint found: *minprice < maxprice*

Impact on Extraction

Cartesian product over all form elements would yield **5328 queries** for extracting all content hidden behind this search interface.

Knowing the constraint identified above, 1,110 meaningless queries would be avoided; over 20% reduction in the number of queries necessary.

Interface



ProFoUnd's Interface Views

1. Browser view – a full-fledged Mozilla-based browser component
2. Constraint view – shows all entry points, conditions, interception points and constraints identified by ProFoUnd
3. Code View – presents relevant HTML and JavaScript fragments for the selection in the constraint view
4. List of URLs
5. Control over highlighted elements
6. Detailed DOM access

Experiments and Results

Experimental Set-up

A total of 70 randomly selected real-estate websites with deep web search interfaces from DIADEM's repository that lists all UK real estate websites made up our data set.

- From the total data set, we manually identified that 30 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 identified all individual integrity constraints enforced on the client-side by hand.

Our aim was to test ProFoUnd's precision when given interfaces with no integrity constraints, but also its combined precision and recall (the default balanced F measure) for interfaces with client-side validation code.

Results

- Score of 1.0 for precision; no false positives identified for all 70 search interfaces
- Score of 0.628571429 for recall
- The harmonic mean was calculated as below, where P is the precision and R is the recall:

$$F_1 = \frac{2PR}{P+R} = \frac{2 \cdot 1.0 \cdot 0.628571429}{1.0 + 0.628571429} = 0.771929825$$

Our F_1 score is very close to the arithmetic mean between the precision and recall indicating a good balance.

Where did we struggle?

- Dealing with obfuscated validation code

```
$('#locInfo').html("Please enter a location.");
```

Obfuscated using JavaScript Obfuscator

```
var _0xa3fb=
["\x50\x6C\x65\x61\x73\x65\x20\x65\x6E\x74\x65\x72\x20\x61\x20\x6C\x6F\x63\x61\x74\x69
\x6F\x6E\x2E","\x68\x74\x6D","\x23\x6C\x6F\x63\x49\x6E\x66\x6F"];$(_0xa3fb[2])
[_0xa3fb[1]](_0xa3fb[0]);
```

- Dynamic creation of objects
- Complex JavaScript features, e.g. *eval*

```
eval("document.forms.psearch.ploc[" + idx + "].checked")
```

- Conditional statements within the scope of *for loops*
- Limitations of the system itself!

Future Work

Moving Forward

To the best of our knowledge, ProFoUnd is the first system to provide integrity constraint identification for search interfaces in the context of deep web extraction based on JavaScript analysis. We have demonstrated how shallow program analysis in this context is a viable solution for integrity constraint identification, achieving good results. However, this is just the first step – there are various future directions to overcome limitations and take this work a step further:

- Moving beyond pattern matching; develop a more generic framework, possibly supported by both supervised and unsupervised machine learning
- Combining static analysis with runtime execution
- JavaScript parsing; Rhino has a number of limitations, we aim to try SpiderMonkey, Mozilla's JavaScript engine
- Utilise error messages better beyond keyword-based error message analysis
- Looking at the server-side for integrity constraints