Discovering User-Defined Event Handlers in Presence of JavaScript Libraries

Shuai Wang, Wensheng Dou*, Chushu Gao, Jun Wei, Tao Huang
Technology Center of Software Engineering
Institute of Software at Chinese Academy of Sciences
Beijing, China
{wangshuai, wsdou, gaochushu, wj, tao}@otcaix.iscas.ac.cn

***Abstract*—JavaScript libraries, such as JQuery, are widely used in web applications. In these libraries' event delegation models, a DOM element's event handler is usually bound to its parent nodes. This makes it difficult for developers to figure out the user-defined event handlers of a specified DOM element. In this paper, we propose an approach that identifies the user-defined event handlers of DOM elements in a web page. We dynamically collect the execution trace for each triggered event in a web page, and analyze how each function is used in the execution trace to discover the event handlers for each event. We evaluate our approach on seven real-world web applications. The result shows that our approach is effective, with an overall precision of 100% and recall of 99.8%.**

Keywords—JavaScript; event; event delegation

I. Introduction

JavaScript libraries, such as jQuery [1], prototype [2], YUI [3], and MooTools [4], are widely used in web applications. These libraries can provide convenient event delegation models, compatibility of different environments, easier integration with other technologies, and so on. The business report [5] shows that 94.8% web applications are built on the top 10 popular JavaScript libraries.

Web applications are event-driven [6]. It is important to know what event handlers can be triggered by an event on a specific DOM element. This information can be used to understand JavaScript programs by developers [7], or automatically generate test cases, such as Crawljax [8].

In order to facilitate the event handling, common JavaScript libraries usually provides an event delegation model. The event delegation model binds the event handlers to the parent element of a DOM element, rather than itself. The event delegation model will dynamically trigger the bound event handlers for the DOM element at runtime. The delegation model hides the user-defined event handlers for each DOM element in a web page. For example, when a user clicks a *div* element, which function is user-defined event handlers for the click event? Existing development tools, such as Chrome DevTools [7], consider the library event handlers (defined in libraries) as the event handlers, and cannot accurately figure out the real user-defined event handlers. Therefore, the user-defined event handlers are hidden by the library event handlers.

It is challenging to accurately identify the user-defined event handlers of DOM elements in a web page. (1) There is no de facto specification for the DOM event delegation model, and different JavaScript libraries implement event delegation in different ways. For example, jQuery implements event delegation using a dispatcher pattern, which uses a single dispatcher function to invoke multiple user-defined event handlers, while YUI using a wrapper pattern, which creates a particular wrapper function for each user-defined event handler. (2) JavaScript is highly dynamic, it treats functions as first-class citizens, it allows developers to create functions dynamically, pass functions as parameters to other functions, return functions as values from other functions, and assign functions to variables, which makes it a particularly difficult language to analyze.

To tackle this problem, we introduce an approach to automatically identify user-defined event handlers of DOM elements. Our approach is inspired by the following observation: user-defined event handlers act like standard event handlers, and are invoked by library event handlers instead of browsers. (1) When event handlers are being bound, functions are passed as parameters to event binding function calls, and are never invoked in the execution of event binding functions. (2) When a corresponding event is fired, an event handler is invoked and receives an argument similar to a standard DOM event [9]. (3) A library event handler may invoke multiple use-defined event handlers in a polymorphic manner. Based on these observations, our approach conducts dynamic analysis on a given web page to discover all the user-defined event handlers. Firstly, our approach figures out candidate user-defined event handlers. We monitor the execution of all functions when the web page is initializing, and check whether a function-type argument is registered as a callback function within the execution of receiver functions, and is not invoked by receivers. Secondly, we trigger all the events on each DOM element and monitor the execution of event handlers. If a candidate event handler is invoked in a polymorphic manner and receives an argument similar to a standard DOM event, this candidate event handler would be a user-defined event handler for the triggered event.

* Corresponding author

In summary, our contributions are as follows:

- We propose an automatic approach to discover user-defined event handlers hidden by JavaScript libraries' event delegation models in web pages.
- We implement a prototype tool for our approach. Our prototype tool allows developers to perform on-the-fly analysis on real-world web pages.
- We perform a systematic experiment on seven real-world web pages, the result shows that our approach can gain high precision and recall.

The rest of the paper is organized as follows. In Section II, we illustrate the motivation and overview of our approach by a real-world example. In Section III, we describe our approach in detail. We show how we have evaluated the approach in Section IV, discuss our work in Section V, and present the related work in Section VI. Finally we give a conclusion in Section VII.

II. Motivation Example

In this section, we introduce a real-world example that will be used throughout the paper. The example is key part of a search engine, and is built on jQuery [1].

A. Example

Fig.1 shows the web page of the example. The page displays three components: a *hot keyword displayer* on the top, an *input receiver* in the middle, and a *keyword recommender* at the bottom. A user can type keywords and click the *Search* button to submit search requests. When a user types keywords, the *keyword recommender*, which is hidden, appears with a list of recommended keywords starting with the input characters. For example, Fig.1 shows the result after typing a character 'a'. The user can hide the *keyword recommender* by clicking on any area of the web page. The *hot keyword displayer* stays on the top all the time, and the user can submit a search request by clicking on a keyword directly. The *search result displayer* component is hidden in the page, at the bottom of the page, under the *keyword recommender*. It is hidden because no search request is submitted yet.

List.1 shows the JavaScript code of the example. It defines six functions. Four of them are user-defined event handlers which are bound by the last four lines of code (lines 30-33). Line 30 binds *search* function to the element with id *submit* as its *click* event handler (line 30), where *search* function constructs a search url (lines 22-23) and asynchronously updates the search results to the web page by *updateSearchResult* function (lines 1-3) via the *ajax* function (line 24). Here, we omit the details of *getById, ajax*, *show* and *hide* functions for the space reason. Line 31 delegates *input*'s *keyup* event handler *showRecommend* on *document.body*. The *showRecommend* function is similar to *search* function, and it asynchronously updates the recommended items by *updateRecommend* function (lines 4-11). Note that *updateRecommend* binds the *click* event handlers to recommended items in the inline mode (lines 7-8), which can be done in an easier way by event delegation, i.e. assigning a

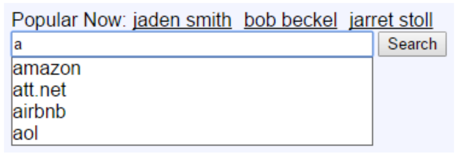

Fig. 1.The Web Page of the Example

```
1   function updateSearchResult(text) {
2     getById('searchResult').innerHTML = text;
3   }

4   function updateRecommend(text) {
5     var keywords = text.split(','), html = '';
6     for (var i = 0; i < keywords.length; i++) {
7       html += '<div onclick="javascript:searchKeyword(event)">'
8                 + keywords[i] + '</div>';
9     }
10    getById('recommend').innerHTML = html;
11  }

12  function showRecommend(event) {
13    var key = getById('input').value;
14    var url = 'http://www.search.net/recommend?key=' + key;
15    ajax(url, updateRecommend);
16    show('recommend');
17  }

18  function hideRecommend(event) {
19    hide('recommend');
20  }

21  function search() {
22    var key = getById('input').value;
23    var url = 'http://www.search.net/search?key=' + key;
24    ajax(url, updateSearchResult);
25  }

26  function searchKeyword(event) {
27    getById('input').value = event.target.innerHTML;
28    getById('submit').click();
29  }

30  getById('submit').onclick = search;
31  $(document.body).on('keyup', '#input', showRecommend);
32  $(document.body).on('click', hideRecommend);
33  $(document.body).on('click', '.keyword', searchKeyword);
```

List 1. JavaScript Source Code of the Example

```
<div class="container">
  <div id="hot">
    <div>Popular Now:</div>
    <div class="keyword">jaden smith</div>
    <div class="keyword">bob beckel</div>
    <div class="keyword">jarret stoll</div>
  </div>
  <div>
    <input type="text" id="input"/>
    <input type="button" id="submit" value="Search"/>
  </div>
  <div id="recommend">
    <div onclick="javascript:searchKeyword(event)">amazon</div>
    <div onclick="javascript:searchKeyword(event)">att.net</div>
    <div onclick="javascript:searchKeyword(event)">airbnb</div>
    <div onclick="javascript:searchKeyword(event)">aol</div>
  </div>
  <div id="searchResult"></div>
</div>
```

List 2. DOM Tree Structure After Typing 'a'

class name *keyword*. Line 32 binds a *click* event handler *hideRecommend* to the *document.body*, but it uses jQuery event

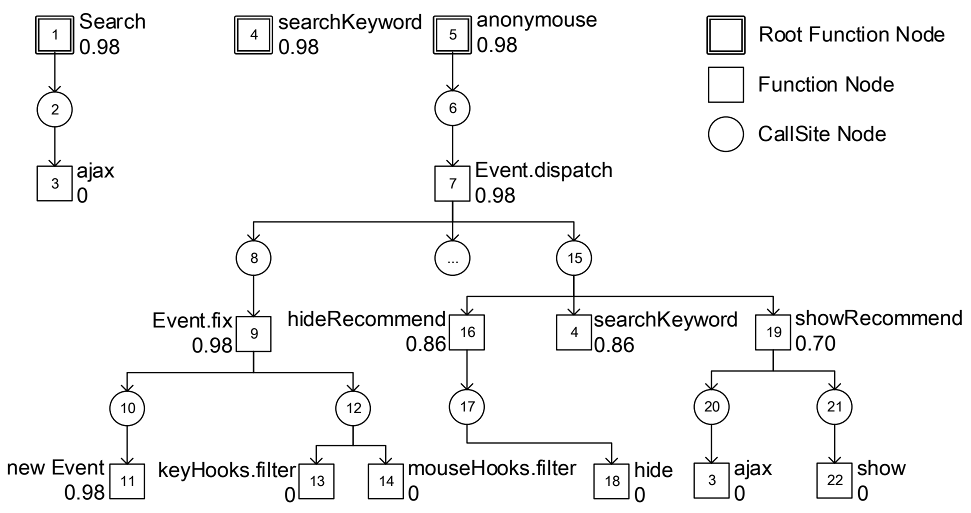

Fig. 2. Merged Call Graphs for the Example

delegation. Line 33 delegates a *click* event handler *searchKeyword* to each child element of *document.body* with the certain class name *keyword*. Note that *searchKeyword* fires *click* event on *submit* via *click*() method.

List. 2 shows the DOM tree structure after typing a character ‘a’. The root *div* element with class name *container* contains all the DOM elements required by the example. The four child *div* elements are corresponding to the four components respectively: the *hot* div is corresponding to the *hot keyword displayer*, the *base* div to the *input receiver*, the *searchResult* div to the *search result displayer*, and the *recommend* div to the *keyword recommender*. Note that the *div* elements within *recommend* div are bound *click* event in the inline mode, while other elements are bound *click* and *keyup* events via function invocations.

When we try to find all event handlers within the page displaying in Fig.1 via developer tools, such as Chrome DevTools [7], we will find the following event handlers:

- Two *click* event handlers are bound to *document.body* (*anonymous function*);
- A *keyup* event handler is bound to *document.body* (*anonymous function*);
- A *click* event handler is bound to the *#submit* (*search*).

Note that event handlers bound by the inline mode, such as *onclick* of *div* elements in List. 2, are not shown.

However, the user-defined event handlers should be:

- A *click* event handler is bound to *document.body* (*hideRecommend*);
- A *keyup* event handler is bound to *#input* (*showRecommend*);
- A *click* event *handler* is bound to the *#submit* (*search*);
- A *click* event handler is bound to each *div.keyword* (*searchKeyword*);
- A *click* event handler is bound to each child node of *#recommend* (*searchKeyword*).

There are many differences between the two lists. We cannot find *hideRecommend* on *document.body*, *showRecommend* on *#input*, and *searchKeyword* on each *div.keyword* from the first list. Instead, we can only find an *anonymous functions* bound on *document.body* for three times, which is actually a library defined event handler in jQuery.

Our example illustrates several important features of JavaScript event delegation model. (1) A library can implement a single event handler, which is used to delegate multiple user-defined event handlers. (2) A user-defined event handler may be implemented as standard event handlers and used in different cases, e.g. it can be used by the library event handlers, or be bound to DOM elements directly, or plays both roles at the same time. (3) The event handler is usually bound to parent elements, so it can be triggered by events fired on any of child elements. Moreover, a library event handler may invoke a user-defined event handler by passing a parameter different but similar to the standard DOM event. For example, a user-defined event handler can receive an argument created by jQuery, instead of the standard DOM event.

B. Challenges

In JavaScript, there is no standard model to clearly define the binding between DOM elements and user-defined event handlers via event delegation. There are several challenges in discovering the binding.

Firstly, there is no de facto specification for the DOM event delegation model, and different JavaScript libraries implement event delegation in different ways. They have different syntaxes, different semantics. For example, the line 31 can be written in MooTools as following:

$(document.body).addEvent(‘keyup:relay(#input)’,showRecommend)

The static check would not work for different libraries.

Secondly, a single event may trigger more than one event handlers, because of *multi-event-binding* and *event-propagation*. In *multi-event-binding*, more than one event handlers are bound to a single DOM element for the same event type, so when an event of that type is fired on the element, all the bound event handlers will be triggered. *Event-propagation* is defined by W3C DOM event model [9], which allows a single event fired on a particular element to propagate

TABLE I. USER-DEFINED EVENT HANDLERS OF THE EXAMPLE

Dom Element	***Event Type***	***Event Handler***	***Kept***
/html/body	click	hideRecommend	Y
//*[@id="container"]	click	hideRecommend	N
//*[@id="hot"]	click	hideRecommend	N
//*[@id="hot"]/DIV[1]	click	hideRecommend	N
[Omit 12 records]			
//*[@id="hot"]/DIV[2]	click	searchKeyword	Y
//*[@id="hot"]/DIV[3]	click	searchKeyword	Y
//*[@id="hot"]/DIV[4]	click	searchKeyword	Y
//*[@id="input"]	keyup	showRecommend	Y

through the DOM tree hierarchy and indirectly trigger other event handlers bound on its parent nodes.

C. Approach Overview

Our approach starts to work as web pages are initializing. Firstly, we monitor how functions are used during page initialization. If a function is registered as a callback function and is not invoked during the execution, it will be regarded as a candidate user-defined event handler.

In the example, after the page initialization and user typing a character 'a', only 4 types of functions satisfy the previous conditions, i.e. *showRecommend*, *hideRecommend*, and *searchKeyword* in callings of function *on*, and *matchers* in calling of function *elementMatcher* defined in jQuery, so the candidate user-defined event handlers are:

$$\{\ showRecommend,\ hideRecommend,\ searchKeyword,\ matchers\ \} \quad (1)$$

Secondly, our approach fires all pre-defined events on each DOM element. During the execution of each triggered event handler, we monitor how each candidate user-defined event handler is invoked. If a candidate event handler is invoked in a polymorphic manner and one of the parameters of the function call is similar to the standard DOM event, it will be regarded as a user-defined event handler for the event. We merge call graphs of all event handlers to identify polymorphic invocations, as shown in Fig.2. It shows three separated call graphs, corresponding to the two event handlers detected by Chrome Dev Tools and the other event handler bound in the inline mode respectively. Note that not all candidates in call graphs are user-defined event handlers, only that invoked by certain call sites are regarded as user-defined event handlers. Our approach generates two-tuples consisting of call site and function as user-defined event handler invocation relationships.

As a result, only three invocations satisfy the above conditions, i.e. node 4 (*searchKeyword*), node 16 (*hideRecommend*), and node 19 (*showRecommend*) are invoked by 15, so the output is:

$$\{\ <15, searchKeyword>, <15, hideRecommend>, <15, showRecommend>\ \} \quad (2)$$

Finally, our approach identifies DOM elements binding user-defined event handlers. We match execution traces with identified invocations, and generate matched combines as the initial result. Note that the initial result may contain a lot of redundancies, because a function may be identified as a user-defined event handler of a child element. We introduce a heuristic to discard redundancies, which will be described in detail in Section III. Table I shows the generated result for the example. Initially, our approach generates 20 records, and 15 of them are redundant. The final result only contains 5 records.

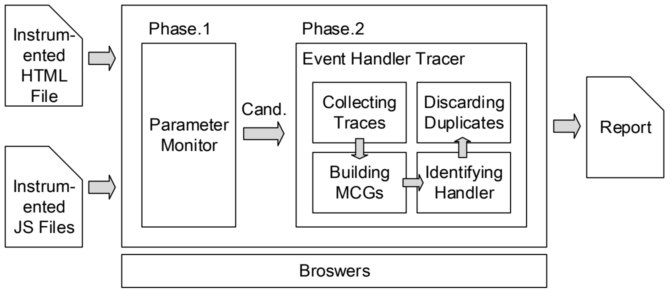

Fig. 3. System Architecture

III. APPROACH

The execution of web pages can be separated into two phases: event-binding phase and event-triggering phase. In the event-binding phase, the web browser initializes the web page, builds presentations of web pages, and binds event handlers to DOM elements. In event-triggering phase, users perform some interactions on web pages, which may fire some events that drive browsers to invoke some user-defined event handlers. Some event handers may generate new presentations and bind new event handlers.

Our approach uses two components to identify the user-defined event handlers. Fig.3 shows the architecture of our approach. Firstly, *Parameter Monitor* monitors the function calls within event-binding phase to check whether the passed parameters act like event handlers. This component receives a web page, which is instrumented by Jalangi [10] automatically, and generates a candidate user-defined event handler set as output. Next, *Event Handler Tracer* monitors the function calls within event-triggering phase to check whether a candidate event handler is invoked like event handlers. It fires all the events in the web page, builds merged call graphs based on collected execution traces of each triggered event handler, identifies user-defined event handlers by applying three heuristics upon the merged call graphs, and finally generate reports after discarding duplicated results.

A. Parameter Monitor

A function could be a candidate event handler if it satisfies the following two conditions: 1) it is passed as a parameter to a function call; 2) it is never invoked during the function call. We check not only on each parameter of function type, but also on properties of each parameter of object type, because libraries may allow developers to bind multiple event handlers within one function call, by passing an object as a parameter which contains more than one event handlers attached as its properties. Such as, jQuery allows developers to pass a single object to bind a *keyup* event handler *showRecommend* and a *click* event handler *hideRecommend* to the *input* element as follows:

```
$(document.body).on('#input',
                    {'keyup': showRecommend, 'click': hideRecommend } );
```

Parameter Monitor performs analysis on each function call. When a function is called, it checks each parameter and each property of parameters of object type. If it finds one of them is a function, it adds a new property to the parameter or the property, to record whether the function is invoked during the execution of current function call. To avoid interfering with execution of subsequent code, the newly-added property is un-enumerable. When the function call is finished, *Parameter Monitor* checks on each function-type parameter or property, if it is not invoked, *Parameter Monitor* decides it is a candidate user-defined event handler.

B. *Event Handler Tracer*

In order to discover all the user-defined event handlers, *Event Handler Tracer* performs the following operations: 1) collecting execution traces for each event handler; 2) merging call graphs; 3) identifying user-defined event handlers by heuristics; 4) discarding duplicated records.

1) Collecting execution traces

Event Handler Tracer first triggers all the event handlers by firing every event in a predefined event set on every element of the web page. *Event Handler Tracer* allows users to specify a collection of events to fire. Now we only fire traditionally user interactive events, i.e. *MouseEvent* and *KeyboardEvent*.

Next, *Event Handler Tracer* collects execution traces by monitoring executions of function invocation expressions, including call sites and invoked functions. We monitor call sites besides functions, because we cannot decide an individual function is a user-defined event handler based on only invocation relationships among functions, and a user-defined event handler may be directly invoked as subroutine of another function. Fortunately, we find that libraries always invoke user-defined event handlers via fixed call sites, so it is necessary to record call sites in execution traces.

Finally, *Event Handler Tracer* generates a call graph for each execution trace as output. The call graph is composited by two types of nodes, *CallSite Node* and *Function Node*.

While recording execution traces, there are some challenges that we have to handle.

Firstly, we aim to collect execution traces of triggered event handlers, which may be disturbed by some other factors, i.e. *asynchronous function execution* and *code-fired events. Asynchronous function execution* includes function invocations triggered by *setTimeout*, *setInterval* and *AJAX* callbacks, while *code-fired events* point to events that are fired not by user actions, but by invocations of particular system APIs, such as *click* and *dispatchEvent*. We find that existing libraries never invoke such methods when invoking a user event handler, so preventing execution of such methods will not prevent execution of user event handlers. Based on this observation, we prevent the invocations of such methods by libraries and user code to collect execution traces.

Secondly, we aim to collect execution traces of individual event handlers, so we must separate multiple event handlers triggered by a single event, just as the second challenge mentioned in Section II. We find that multiple event handlers triggered by the same event are invoked by system calls, they will never co-exist in a call stack at the same time, so we record the function call stacks, and identify an invoked function as an individual event handler when the call stack is empty.

2) Merging call graphs

User-defined event handlers should be invoked in a polymorphic manner. Currently, *Event Handler Tracer* uses merged call graphs (MCG) to identify polymorphic invocation style, i.e. a polymorphic invocation is a single call site invoking multiple functions in MCG.

Event Handler Tracer firstly groups execution traces by their root function entrance, and builds a single MCG for each group.

Event Handler Tracer builds MCGs as follows: firstly, it creates a root *Function Node* for each group, and then iteratively scans each execution trace of the group. When it meets a *CallSite Node*, it creates a new *CallSite Node* if it does not exist in current MCG, and adds an edge from current *Function Node* to this *CallSite Node*. When it meets a *Function Node*, it creates a new *Function Node* if it does not exist in current MCG, and adds an edge from current *CallSite Node* to this *Function Node* if there is no such edge. For example, in Fig.2, *Function Nodes* 4, 16, and 19 are invoked by *CallSite Node* 15 in separated execution traces initially, and are merged to be invoked by a common *CallSite Node* finally.

Finally, we get a set of directed cyclic graphs as MCGs, which will be used for identifying user-defined event handlers.

3) Identifying user-defined event handlers

Based on candidate event handlers generated in *Parameter Monitor*, and the MCGs, *Event Handler Tracer* identifies user-defined event handlers by the follow heuristics:

H1: A user-defined event handler is invoked by a call site which invokes more than one functions.

H2: A user-defined event handler is a function that is invoked receiving a parameter similar to the standard DOM event.

We use H1 to check whether a candidate event handler is invoked in the polymorphic manner, which means the candidate is more likely to be a user-defined event handler. Jensen et al. [11] found that most invocations in the web applications are monomorphic, based-on this observation, H1 discards a lot of false positives from candidate event handlers.

We compute the possibility of a candidate being an event handler based on H2. We believe that event handlers are defined to handle events, so they should always be defined receiving an event-type argument. However, the weakly-typed nature of JavaScript allows developers to define an event handler receiving no argument if there is no need to get information from the event object, so do a lot of developers.

TABLE II. EXPERIMENTAL RESULT

Subjects	Library	NE	LOC	NA	NF	NC	NH	NU	NH/NU	Time(s)
www.ask.com	jQuery	261	8,278	9	9	9	9	116	7.8%	4.8
www.about.com	jQuery	4,473	36,291	199	199	199	28	1,480	1.9%	154.9
www.chess.com	jQuery	549	27,733	2	2	2	2	1,065	0.2%	22.2
www.fool.com	Prototype	640	18,248	6	6	6	6	363	1.7%	14.1
www.wordreference.com	YUI	261	3,056	6	6	6	6	42	14.3%	1.1
www.mid-day.com	jQuery	1,973	39,636	235	235	235	82	2,233	3.7%	566.4
www.zoominfo.com	jQuery	645	19,355	1	0	0	-	-	-	38.6

NE: Number of DOM Elements, LOC: Lines of JavaScript Code, NA: Number of Actual User-Defined Event Handlers, NF: Number of Found User-Defined Event Handlers
NC: Number of Correctly Found User-Defined Event Handlers, NH: Number of User-Defined Functions as Event Handlers, NU: Number of User-Defined Functions

Even so, libraries should always pass a parameter of event type to event handlers. Similarly, the weakly-typed nature allows libraries to pass a different parameter but similar instead of the standard DOM event object, such as *DOMEvent* passed by MooTools. It is challenging to determine whether an invoked function serves an event handler.

To handle this challenge, *Event Handler Tracer* computes the similarity of parameters with standard DOM event, to illustrate the possibility of invoked function served as an event handler. In detail, we compute the similarity of properties of each parameter with properties of standard event object. An parameter owning more same properties with standard event object gains higher possibility. Now we define the standard *MouseEvent* properties as that defined in W3C defined DOM level 2 event model, while the standard *KeyboardEvent* properties as the actual standard which is followed by mainstream browsers, because W3C DOM level 2 event model does not define the *KeyboardEvent* interface. We identify a candidate event handler being an event handler if the possibility is higher than a threshold. Currently, the threshold is set to 0.2, which is gained from our experiments. We find that 0.2 is able to identify almost all of the non-event-type object, and indicates that some libraries implement only popular properties of event in their own event type at the same time.

A candidate event handler, which is invoked as an event handler (H2), in polymorphic manner (H1), is identified as a user-defined event handler.

However, this process may cause false positives. Some libraries implement event delegation using wrapper pattern, which create a wrapper function for each user-defined event handler. Invocations of these wrapper functions may satisfy our previous conditions too. To reduce these false positives, we introduce the third heuristic to discard wrapper functions of event handlers.

H3: If an identified user-defined event handler invokes another user-defined event handler satisfying H1 and H2, then it should be a wrapper.

4) Discarding duplicated records

The result generated so far contains a lot of duplicated records, because event propagation allows that events fired on child elements to trigger the event handlers bound to their parent elements, so a user-defined event handler is identified as being bound to the child elements of the original one, which results in a lot of redundancies.

We introduce a heuristic to discard the duplicated event handlers. Since the event fired on a child element also triggers event handlers bound to its parent element, we decide an identified event handler is duplicated if it is also bound to its parent element, and remove it from the final report.

Note that the above method is unsound. Our approach discards the same event handlers bound to child elements if they are bound to the parent element, which may exist in real world. Developers can stop propagation of events by calling a '*stopPropagation*' function, and bind a same event handler to child elements and their parent at the same time. In this situation, our approach may discard event handlers bound to child elements. However, we believe that this unsoundness is acceptable, for simply removing the event bound to child elements and the '*stopPropagation*' expression within the event handler function generates the same result as the original one, and this is exactly the same as what our approach does.

IV. EVALUATION

We selected 7 real-world web applications and ran our tool on them. We manually checked discovered user-defined event handlers. We try to answer the following research questions:
RQ1: Is our approach accurate in detecting user-defined event handlers?
RQ2: How quickly can our approach perform the user-defined event handler detection analysis?
RQ3: What is the effect of our approach on detecting user-defined event handlers?

A. Experimental Subjects and Setup

For our empirical evaluation, we selected 7 web applications from Alexa top sites [12]. The selected web applications are from different categories, built with different libraries, of different code sizes, and of different popularities. Table II provides the basic information of the 7 applications we selected. The Column Subjects shows the URL of selected web applications, Library shows the library they use, NE shows the number DOM elements contained in the web page, and LOC shows the number of lines of JavaScript code of the page. Note that 5 out of 7 web applications use jQuery, which is consistent with the reality. Some subjects use minimized version of libraries, we formatted the minimized source code before counting the lines of source code.

We deployed a proxy server to perform the on-the-fly instrumentation for subject systems. In order to perform the experimental sequentially, we implemented a plugin that prevents JavaScript code to modify the state of browsers, so we can fire events on a fixed web page again and again. We ran all the analysis on a Google Chrome browser, on a computer with Intel Core i7 CPU 3.40 GHz processer, 8GB of memory. We performed analysis on the home page of each selected web applications for 10 times, and computed the average values as the final results.

B. *Results*

1) RQ1: To answer RQ1, we saved web pages under testing to local storage, and analyzed the saved source code manually to discover the actual user-defined event handlers. The number of actual user-defined event handlers are shown in the fifth column (NA) of Table II. After that, we compared the analysis report generated by our approach with that by our manual work to determine the correct results from the analysis report. The sixth column (NF) shows the number of user event handlers found by our approach, the seventh column (NC) shows the number of correct results the approach found. The experimental result shows that our approach is able to discover user-defined event handlers for web pages. Our approach can gain overall 100% precision and 99.8% recall.

There is only one user-defined event handler that has not been found. It is the only user-defined event handler for event delegation in *zoominfo*. Because our approach determines polymorphic invocations by the existence of one call site invoking multiple functions, not able to determine one call site invoking only one function as a polymorphic invocation, so our approach cannot discover the only one user event handler within a web page.

2) RQ2: To answer RQ2, we just recorded the execution time of each analysis job. The eleventh column (Time) of Table II shows the average execution time for each subject system. From the table we can find that for some web pages with less than 500 DOM elements and using less than 10 thousands lines of JavaScript code, our approach is able to finish the analysis job within a few seconds; for web pages with less than 1,000 DOM elements and using less than 30 thousands lines of JavaScript code, it will finish the analysis job within tens of seconds; only for web pages with large scale of DOM elements and using large scale of JavaScript, our approach costs more than one minute to finish the analysis job. However, we believe the time cost is acceptable.

3) RQ3: To answer RQ3, we introduce a factor to show how many user-defined functions are filtered out as event handlers. We compute the factor by dividing the number of user-defined functions as event handlers by the number of user-defined functions. A lower factor means fewer user-defined functions are registered as event handlers.

We manually counted the number of user-defined functions (NU) and that registered as event handlers (NH), and computed the factor by dividing NH by NU, as shown in the eighth, ninth and tenth columns of Table II. Note that a function is allowed to be bound to multiple DOM elements, so NH may less than NF in the same row in Table II. The result shows that user-defined event handlers only take a small portion of all user-defined functions, from 0.2% to 14.3%, with an average 2.5%, so our approach can be greatly helpful in detecting user-defined event handlers.

C. *Threats to Validity*

Our evaluation is performed on only 7 web applications. However, these web applications considered are selected from Alexa top sites, and use different JavaScript libraries. They can represent the main web applications. Further, we manually identified the user-defined event handlers in these web applications, and we may miss some event handlers. But, we have done our best to identify all the event handlers. In future work, we will evaluate our approach on more web applications with more JavaScript libraries.

V. Discussion

Generally, libraries invoke all of the user-defined event handlers via a shared function call expression, which presents as a one-to-many invocation relationship. Our approach levarages this one-to-many invocation relationship as one of the conditions to identify user-defined event handlers. However, in some extreme use cases, there may be only one user event handler in a single web page, when the one-to-many relathionship will not appear, such as the home page of *www.zoominfo.com* showing in section IV. Our approach is not able to handler such web pages with single user event handler. We try to idenfity such single user event handler in our future work.

Moreover, we believe that event handlers should handle events, which should be invoked passing a parameter similar to the standard DOM event, just as the W3C event model. However, there are some libraries that encapsulate the standard DOM event objects and create almost complete different objects, such as KnockoutJs [13], a open source library that supplies dynamic JavaScript UIs with the Model-View-View Model (MVVM) pattern. The MVVM pattern is quite similar with MVC pattern, where KnockoutJs plays the role of Controller, and developers are required to implement couples of Model and View. When an event fires, KnockoutJs creates a new Model object based on the fired event, and passes the Model as a parameter to corresponding event handler. Our approach is not able to handle web pages built on such deeply-encapsulating-event libraries, we will handle them in the future work.

VI. Related Works

Web Application Comprehension. Alimadadi et al. [14] presents a tool called Clematis to help developers in understanding the complex dynamic behavior of web applications. Clematis monitors the execution of JavaScript while developers are interacting with web pages, and displays the execution trace and effects at three different semantic levels of granularity. While similar to our work, Clematis maps DOM elements to event handlers, it only maps event handlers triggered by user actions, without distinguishing whether the event handlers are bound to target elements directly or not, and it records all executed functions, without identifying user-defined functions. Our work identifies user-defined event handlers functions and their bound elements.

UI Feature Location. Maras et al. [15] [16] presents a browser plugin called FireCrow to derive the implementation of UI features on the client side. FireCrow leverages dependency graphs among HTML, CSS and JavaScript to identify code related to certain features. Li et al. [17] present a tool called Script Insight to locate the implementation of a particular feature of UIs in JavaScript code. Script Insight builds a mutation of dependency graph between DOM and JavaScript, for identifying JavaScript code performing certain mutations on DOM. FireCrow and Script Insight focus on mapping JavaScript expressions to given UI features, while our work focuses on mapping user-defined JavaScript functions to DOM elements.

JavaScript Call Graph. Call Graph is used in many fields. Madsen et al. [18] leverages call graph to help type inferring for JavaScript. Feldthaus et al. [19] presents methods to build JavaScript call graphs efficiently for IDE services. Jensen et al. [10] builds call graphs for modeling the HTML DOM and browser APIs. These works focus on handle problems occurred in client-side code. Nguyen et al. [20] focuses on client-side code embedded as string literals within server-side code, they build conditional call graphs for embedded HTML, CSS, and JavaScript. All these work aims to resolve problems occurred in developing phase using static analysis, while our work aims to help developers to understand existing web applications using dynamic analysis.

Web Application Debugging. Record and replay is widely used to reproduce web application bugs [21] [22]. Such methods usually record all the events to ensure faithful replay of bugs. This results in a lot of events that trigger no execution of user-defined event handlers. Our approach can be helpful in reducing such redundant events for record and replay for web applications.

VII. Conclusion

User-defined event handlers are often hidden by library event handlers when using JavaScript libraries in web pages. We propose an approach that is able to identify user-defined event handlers automatically. Our approach fires all the events on a web page, and checks whether each invoked function is bound and invoked like an event handler, and the most similar ones are regarded as user-defined event handlers. The evaluation on seven real-world web pages shows that our approach can identify user-defined event handlers with great precision and recall.

Acknowledgments

We sincerely thank Jie Wang for her valuable comments and careful proofreading. The work was supported in part by National Grand Fundamental Research 973 Program (2015CB352201) and National High Technology Research & Development 863 Program (2013AA041301) and National Natural Science Foundation (61379044) and National Key Technology R&D Program (2015BAH55F02) of China.

References

[1] “jQuery.” [Online]. Available: http://www.jquery.com.

[2] “prototype.” [Online]. Available: http://www.prototypejs.org.

[3] “YUI.” [Online]. Available: http://www.yuilibrary.com.

[4] “MooTools.” [Online]. Available: http://www.mootool.net.

[5] “Builtwith.” [Online]. Available: http://www.builtwith.com.

[6] G. Li, E. Andreasen, and I. Ghosh, “SymJS: Automatic Symbolic Testing of JavaScript Web Applications,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 449-459.

[7] “Chrome DevTools. ” [Online]. Available: https://developer.chrome.com/devtools.

[8] “Crawljax.” [Online]. Available: http://crawljax.com/.

[9] “Document Object Model Events. ” [Online]. Available: http://www.w3.org/TR/DOM-Level-2-Events/events.html.

[10] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, “Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript,” in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2013, pp. 488-498.

[11] S. H. Jensen, M. Madsen, and A. Møller, “Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications,” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)*, 2011, pp. 59-69.

[12] “Alexa.” [Online]. Available: http://www.alexa.com/topsites.

[13] “KnockoutJs.” [Online]. Available: http://www.knockoutjs.com.

[14] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman, “Understanding JavaScript Event-Based Interactions,” in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014, pp. 367-377.

[15] J. Maras, J. Carlson, and I Crnkovic, “Extracting Client-side Web Application Code,” in *Proceedings of the 21st International Conference on World Wide Web (WWW)*, 2012, pp. 819-828.

[16] J. Maras, M. Stula, J. Carlson, and I. Crnkovic, “Identifying Code of Individual Features in Client-side Web Applications,” in *The IEEE Transactions on Software Engineering (TSE)*, Volume:39, Issue: 12, pp. 1680-1697, Dec. 2013.

[17] P. Li and E. Wohlstadter, “Script InSight: Using Models to Explore JavaScript Code from the Browser View,” in *Proceedings of the 9th International Conference on Web Engineering (ICWE)*, 2009, pp. 260-274.

[18] M. Madsen, B, Livshits, and M. Fanning, “Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries,” in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2013, pp. 499-509.

[19] A. Feldthaus, M. Schäfery, M. Sridharanz, J. Dolbyz, and F. Tip, “Efficient Construction of Approximate Call Graphs for JavaScript IDE Services,” in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 752-761.

[20] H. V. Nguyen, C. Kästner, and T. N. Nguyen, “Building Call Graphs for Embedded Client-Side Code in Dynamic Web Applications,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 518-529.

[21] J. Mickens, J. Elson, and J. Howell, “Mugshot□: Deterministic Capture and Replay for JavaScript Applications,” in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2010, pp. 159-174.

[22] J. Wang, W. Dou, C. Gao, and J. Wei, “Fast Reproducing Web Application Errors”, *in the 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015.