Finding and Emulating Keyboard, Mouse, and Touch Interactions and Gestures while Crawling RIA's

Frederik H. Nakstad, Hironori Washizaki, Yoshiaki Fukazawa
Department of Fundamental Science and Engineering
Waseda University
Tokyo, Japan
frederik@fuji.waseda.jp

***Abstract*—Crawling JavaScript heavy Rich Internet Applications has been a hot topic in recent years, giving us automated tools for indexing content, test generation, and security- and accessibility evaluation to mention a few examples. However, existing crawling techniques tend to ignore user interactions beyond mouse clicking, and therefore often fail to consider potential mouse, keyboard and touch interactions. We propose a new technique for finding and exercising mouse, keyboard, and touch interactions when crawling highly interactive JavaScript-based websites by analyzing and exercising event handlers registered in the DOM. A basic form of gesture emulation is employed to find states accessible via swiping and tapping. Testing the tool against 6 well-known gesture libraries and 5 actual RIA's, we find that the technique discovers many states and transitions resulting from such interactions. Our findings indicate the technique could be useful for automatic test generation, error discovery, and accessibility evaluation, especially for mobile web applications with advanced interaction options.**

Keywords-crawling; gesture emulation; event handler analysis; RIA

I. INTRODUCTION

Web applications have become more and more advanced over the past few years with the maturation and increased adoption of HTML5 and its related API's. As a result of this, the amount and complexity of JavaScript code on the client-side has grown, giving us a different breed of web application capable of much more advanced functionality and richer user interaction models than before. Compounding this is the advent of the smartphone, which has popularized touch screens and made various gesture interactions part of most people's technical vocabulary. A recent report finds that users in the US now spend more time consuming media using their mobile and tablet devices than desktop computers [8].

This rise in client-side complexity coupled with the dynamic nature of JavaScript has introduced many challenges in how to reliably crawl such web applications. While traditional websites rely on anchor tags and buttons for navigation and are static in terms of content on the client-side, new JavaScript-reliant RIA's can change dynamically and drastically as the result of JavaScript manipulating the DOM of the page without the need of a round-trip to the server for new HTML. In papers such as [1, 7] methods for automatic crawling of JavaScript reliant RIA's have been introduced. These methods have spawned a variety of applications to automate beneficial tasks such as indexing for search engines [1], accessibility and usability evaluation [6, 16], automatic test generation [3, 11, 12, 13], regression testing [4], and security testing [5] to mention some.

One aspect oft neglected is the fact that this new breed of web applications are capable of very advanced interactions. Existing research focuses heavily on simple interactions initiated by mouse clicks, usually ignoring other keyboard, mouse, and touch events, not to mention gestures. Another common assumption is that most state transitions can be reached by interactions with <a>, <button>, and <input> elements when choosing candidate interactions as often done by empirical studies such as [10]. This might be okay for web sites containing simple interaction models on this small subset of elements, but we believe many states and transitions could be missed for RIA's with more advanced UI and mobile web applications.

In this paper we propose a set of extensions to Crawljax with functionality enabling it to detect the event handlers available in each state. These detected event handlers are then used as a basis for identifying new candidate interactions,. The candidate interactions are exercised using programmatic event construction and gesture emulation, leading to an increase in discovered states and transitions.

The following research questions are addressed:

- **RQ1.** How comprehensively and efficiently can our technique capture and perform gesture interactions?
- **RQ2.** How often do various keyboard, mouse, and touch events lead to new states in modern RIA's, and which event types are more likely to induce new states?
- **RQ3.** What DOM elements are more likely to be targets for interactions leading to new state transitions?

The following contributions are offered:

- A technique for discovering all event handlers of a state when crawling websites.
- A set of extensions to Crawljax enabling it to identify candidate transitions based on event handler analysis and emulate simple event dispatches as well as tap and

(DOI reference number: 10.18293/SEKE2015-059)

swipe gestures. Henceforth this modified version of Crawljax is referred to as mobCrawler[1].

- Evaluation of mobCrawler's ability in emulating common gestures and interactions by testing against 6 of the most popular gesture libraries for JavaScript available.
- A case study against 5 modern RIA's with advanced interaction models evaluating the efficiency and usefulness of event handler analysis when crawling advanced RIA's.

II. BACKGROUND

A. Crawling RIA's and Crawljax

Crawling websites used to be relatively less complex than it is today. For traditional websites the content of each page is static after it has been loaded in the browser, and other static pages were loaded by following anchor or button elements. In modern rich Internet applications using JavaScript things are more complex. Changes to the DOM can come as the result of asynchronous HTTP requests loading new content, or event handlers and timeout events firing custom code. Additionally, the dynamic nature of the JavaScript language itself can provide flexibility as well as unintended states and side effects [2].

Crawling such applications is usually done by loading a starting page, its DOM recorded as the initial state, and then automatically exploring the various interactions possible on the page to elicit changes in the DOM. Each interaction causing a change is recorded as a transition, and each modified DOM is recorded as a new state.

One of the most prominent tools fulfilling this purpose is Crawljax. Crawljax is a highly customizable crawler aimed at JavaScript-heavy web sites. It performs a depth-first search of the target web site using the Selenium Web Driver to control the browser, and has many customization options for things such as setting crawl depth, state abstraction comparators, and crawl rules for ignoring certain links. It performs a depth-first search trying to detect as many states and transitions as possible within the constraints specified by the user. Due to this open and customizable nature we implement our technique as a set of modifications and extensions to Crawljax.

B. Event Handler Registration

When adding event handlers to a web page, there are three options available through the browser API:

1. Programmatically use a target elements addEventListener() function
2. Programmatically use a target elements on[eventType] attribute, [eventType] indicating the type of event handler
3. Declare a on[eventType] attribute on the target elements HTML tag, [eventType] indicating type of event handler

There are countless libraries offering other API's to attach event handlers, but they all eventually resolve to one of these standardized, "native" API calls offered by the browser. Using any of these options, developers are able to attach custom code to be fired when one of a multitude of event types is performed on the given element. The code in these event handlers may manipulate the DOM, potentially eliciting a new state. Gestures are usually developed as a sequence of related event handlers on event types such as touchstart, touchmove, and touchend, analyzing each event sequence's properties to determine what gesture was performed.

Though a common pattern is to attach event handlers once the DOMContentLoaded event of the browser has fired as part of the page's lifecycle, developers can technically add events as soon as JavaScript is being evaluated by the browser. New event handlers can also be attached at any point after page load, as part of other event handlers being exercised.

C. Mobile Devices

According to [9] web browsing on mobile and tablet devices has increased rapidly the last few years, and together constitutes 37.48% of all usage. As mobile navigation of the web increases, developers may also want to adapt their website to take advantage if touch screen input and gestures. This could mean that many states and transitions not reachable by simple mouse clicks anymore. Previous techniques may miss important states relying on touch, mouse, and keyboard interactions, especially for mobile web applications relying on gestures. We believe actually exercising such interactions will help in finding new states and transitions whether it is for indexing content, automatically evaluating accessibility, or employing automated testing approaches for the target web application.

There has been a huge increase in use of the web from smartphones [9], and as a result of this many web applications also offer an especially tailored mobile version. These mobile versions often take advantage of the devices touch screen, and may implement certain navigational interactions using touch and gestures. It therefore seems likely to us that such web applications may contain states and transitions only reachable through touch and gesture interaction.

```
<div id="gallery">
  <img class="image" src="mtfuji.jpg" />
  <label class="caption">Mt. Fuji</label>
</div>

$('#gallery').swipeLeft(function() {
  // Load new image and caption via AJAX
  ...
});
```

Figure 1. Gesture Handler Registration

[1] https://github.com/fnakstad/mobcrawler

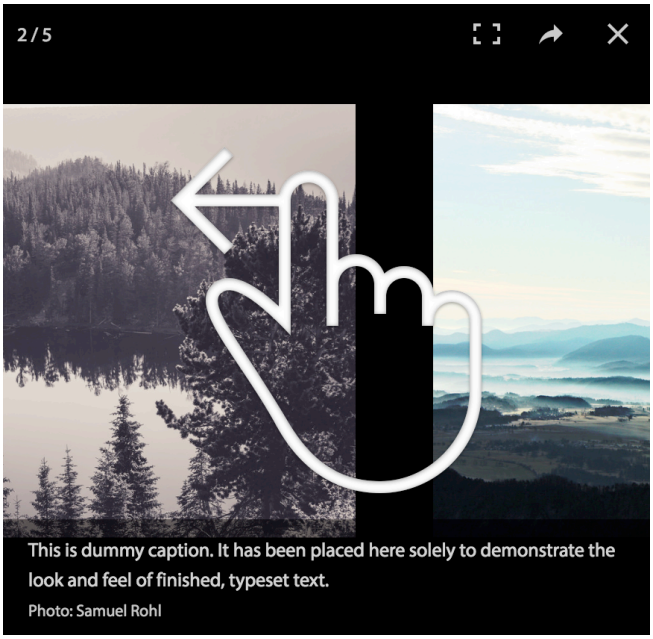

Figure 2. Gallery with swipe interactions

D. *Motivating Example*

For our motivating example we consider a photo gallery component embedded on a web site containing an image and some descriptive text as pictured in Figure 1 and 2. In order to load the next picture and text caption in the gallery you have to swipe left or right. These swipe interactions would likely be implemented by attaching the desired gesture type to a target DOM element as seen in the code sample. Since existing crawl techniques don't attempt to execute such advanced interaction events, the states loaded by performing these swipe gestures would not be found, leaving a good chunk of application functionality and content unexplored. This kind of interaction pattern has become quite common and can often be seen in photo galleries, carousels for news stories, especially featured sales products, and so on.

III. Event Handler Analysis and Gesture Emulation

A. *Detecting Registered Event Handlers for a State*

In order to find out what event handlers have been registered in any given state it is necessary to instrument the native addEventListener() function implementation to keep track of all event registrations performed by the target web site's JavaScript code. It is important that our instrumentation code is performed before any other client-side code in order to catch all events programmatically added to the page. This means we need to inject this instrumentation module at the very top of the documents <head> tag before the page is loaded in the browser. This approach ensures that any event handlers attached during page load as well as dynamically when crawling to another state from initial page load will be detected by our tool.

```
var original =
  EventTarget.prototype.addEventListener;
EventTarget.prototype.addEventListener =
function() {
  var type = arguments[0];
  addEventHandler({
    type: type,
    xpath: getElementXPath(this),
  });
  original.apply(this, arguments);
};

window.addEventListener('load', function(){
  walkSubtree(document);
},false);
```

Figure 3. Event Handler Registration Detection

There may also be event handlers attached directly via on[eventType] attributes on the DOM elements. These event registrations are not processed through addEventListener(), so we need to handle them separately. After the page is loaded we walk the entire DOM tree, polling each element for any such event handler registrations.

As seen in the code snippet in Figure 3, The addEventHandler() function will check whether the event type is of a type we are interested in, which can be configured in the script, and if so add it to a list. Once the DOM is loaded, the script will call the walkSubtree() function on the document object, which will recursively traverse the entire DOM tree looking for event handlers attached via the on[eventType] attribute, and add them via addEventHandler(). This means we can effectively monitor any and all event handler registrations performed on the page.

B. *Injecting Script via a MITM Proxy*

Figure 4 shows the structure of mobCrawler. Neither the browser itself nor the Selenium WebDriver used by mobCrawler to control the browser offers a way to manipulate the HTML before it is evaluated. Therefore we inject the instrumentation script using a faux man-in-the-middle attack via a proxy server.

This proxy server is implemented using the well-known mitmproxy [14] program, and run with a custom script. The proxy server sniffs for any relayed HTML content, intercepts it, and modifies the <head> tag to inject our JavaScript module. The injected JavaScript module then attaches one non-intrusive JavaScript object on the window object, which can be configured to listen for event handler registrations of any specified event types. mobCrawler can communicate with this module via JavaScript operations executed in the browser to fetch the current event handlers for a state as well as exercise various interactions on the web page. Using a man-in-the-middle proxy server poses some problems with regards to the browser denying 3rd party SSL certificates for certain sites or forbidding functionality which might be restricted due to CORS security policies. A preferable option would be to use the WebDriver or web browser itself to inject this script into the document before evaluating it, however this is not supported as of today. We configure mobCrawler to use this proxy server when crawling.

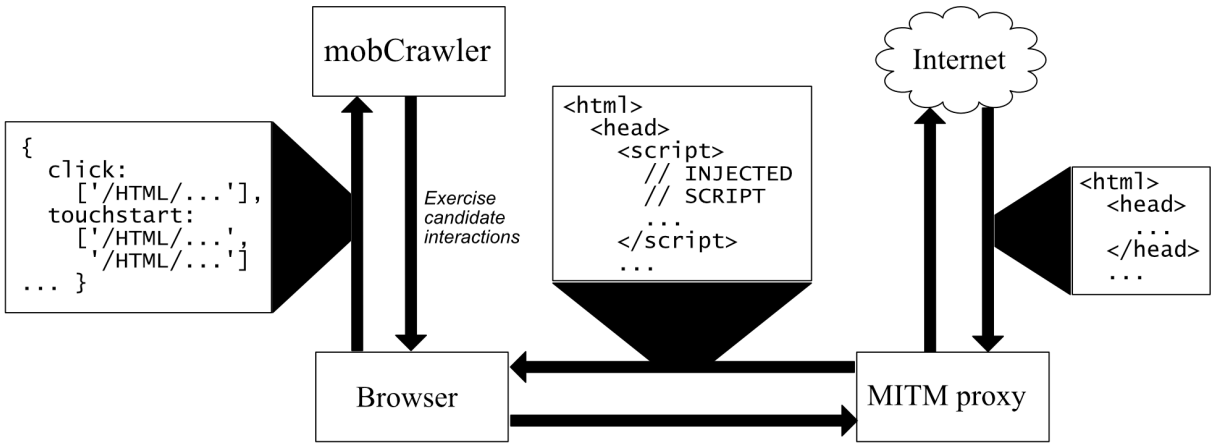

Figure 4. Structure of mobCrawler

TABLE I. GESTURE EMULATION

Interaction	Description	Fire On
Event dispatch	Create and dispatch an event to the target element	click, dblclick, mouseover, mouseout, keydown, keyup, keypress
Mouse swipe	Emulates a mouse swipe in a given direction (left, right, up or down), by creating and dispatching a series of mousedown, mousemove, and mouseup events.	mousedown
Touch swipe	Emulates a touch swipe in a given direction (left, right, up or down), by creating and dispatching a series of touchstart, touchmove, and touchend events.	touchstart
Tap	Emulates a tap by dispatching touchstart and touchend events with a small timeout.	touchstart
Double tap	Two tap events in quick succession.	touchstart
Tap hold	Same as a tap but with a longer timeout between the touchstart and touchend events.	touchstart

C. *Emulating Browser Interactions*

mobCrawler can now fetch all registered event handlers registered in a state by calling our injected JavaScript module. This list of event handler registrations will in turn form the basis for what candidate elements and interactions we choose to perform. Though we can possibly detect and emulate all possible browser event types, we choose to focus on mouse, keyboard, and touch events. For some of these events, such as mousehover, mouseout or keydown it is sufficient to create and dispatch a corresponding event object to the target element. These events are created via the JavaScript module injected into the page, and will programmatically construct events of the desired type mirroring the W3 specifications in [17]. The parameters set will be based on the target element in question and what the desired interaction specified is, then finally dispatched to the target element in question.

For keyboard events there is a big span of keys that can be set to be activated. In our simulations we use the somewhat naïve approach of always setting the key being pressed to the character 'e'. It seems like many more states can be elicited with a more advanced strategy of when to use what characters. Many websites include various keyboard shortcuts which could increase the number of found transitions and states if exercised. However we do not consider that beyond our rather simple approach in this paper.

Additionally, we emulate certain common gestures using mouse and touch events as specified in Table I. Variations of tap gestures are only considered for touch, while swipe gestures are also implemented for mouse. In this paper we only consider single-touch gestures, ignoring more advanced gestures involving more fingers. We do this since we find it less likely they will induce meaningful states and to reduce the number of candidate interactions performed.

Swipe gestures can be emulated in 4 different directions, and are calculated by starting at the center of the target element, then rapidly being moved 400 pixels in the given direction. Candidate interactions are created based on the event handler registrations detected in the current state. This means that if a touchstart event handler is found on an element, a total of 7 candidate interactions will be attempted: swipe up, swipe down, swipe left, swipe right, tap, double tap and tap hold. This can lead to a rapid increase in candidate interactions, but can be customized to add only a subset of these candidate interactions. For example, we could instruct mobCrawler to only try swipe right and single tap interactions for elements containing touchstart event handler registrations if that is desired.

Using this approach we can see how the problem posed in the motivating example can be solved. The JavaScript module injected into the browser via the MITM proxy will detect a touchstart event registered on the gallery element, and record it in a list of event handler registrations. Once the page has loaded mobCrawler will fetch this list of event handler registrations, and process them in turn. When it reaches the touchstart event registered on the gallery element, it will attempt various swipe and tap gestures as described in Table I to see if the action changes the state of the DOM. Once it tries swiping left and swiping right the content inside the gallery should change, and mobCrawler will register these new transitions and states in the state graph.

TABLE II. GESTURE SUPPORT

Library	Tap	Double tap	Tap hold	Swipe			
				Left	*Right*	*Up*	*Down*
Hammer.js	✓	✓	✓	✓	✓	✓	✓
Quo.js	✓	×	×	×	×	×	×
dojox.gesture	✓	✓	✓	✓	✓	✓	✓
touchSwipe	✓	✓	✓	✓	✓	✓	✓
Touchy	×	×	×	×	×	×	×
jGestures	✓	N/A	N/A	✓	✓	✓	✓

D. Mobile Websites

It has become increasingly common for websites to offer a separate mobile version, which can be very different from the plain "desktop" version. In many cases the mobile version is a completely separate implementation from the desktop version to create a user experience entirely tailor made for mobile and tablet devices. The most common technique used by websites to differentiate between mobile and desktop users is user agent sniffing, which will parse the user-agent string in the initial HTTP request, and then serve back either the mobile or desktop version of the website depending on the device reported in the user agent string.

We add functionality to mobCrawler which enables the user to imitate a mobile device by overriding the user-agent string sent in HTTP requests to the webserver of the site we are crawling. If the user wants to imitate a mobile device, this string can be set to the user agent string reported by a browser built for the iOS or Android platforms. Some gesture libraries and websites will also offer different functionality based on whether the browser reports the user's device as being touch enabled or not. For such cases, we inject a polyfill spoofing that we support touch even if we are crawling using a non-touch device.

IV. EVALUATION

A. Gesture Dection and Emulation (for RQ1)

To evaluate the effectiveness of emulating gestures, we set up an experiment where we task mobCrawler with finding gestures as implemented by various popular JavaScript gesture libraries. Though it is possible to create gestures from scratch just by using the native touch and mouse events, it seems likely to assume that the majority of developers will rely on a gesture library to make their jobs easier. This also gives us the opportunity to investigate many different implementation approaches, and seeing if our tool can handle them.

We create a basic website for each library with a element on which we register gesture interactions using the various libraries. Though many of the libraries offers a big set of gestures we focus only on simple single-point gestures as described in Table I.

As can be seen in Table II, the approach was able to handle the gesture interactions of most libraries flawlessly with two exceptions. Both Quo.js and Touchy create and register custom event types, which they use when registering their gestures. Since our code instrumenting addEventListener() registrations was configured to only look for a predefined set of standard event types it didn't pick up on these non-standard event type registrations. However, mobCrawler can be configured to also look for these custom events, and fire corresponding gesture interactions on them to expand library support.

The majority of gesture libraries seem to use the native touchstart and mousedown events, and in these cases our tool was able to exercise all the transitions and find all the given states. Note that jGestures did not have API's for registering doubletap and tap hold gestures, and so those gestures are not considered. Hammer.js adds both mouse and touch handlers for gestures, while Quo.js and dojox.gesture actually checks mosue and touch capabilities of the device before registering the appropriate event handlers.

A drawback of the approach is the large number of candidate interactions which have to be exercised in order to find valid states. Since we can't make any assumptions about what gesture interaction is required to elicit a new state, we resort to exercising all of them. For example, upon finding an element with a touchstart event handler attached, we create candidate elements for all 3 tap gestures as well as swipes in 4 different directions. This might lead to a high recall of new states, but precision is lacking and can lead to a lot of failed candidate interactions. Reducing the candidate gesture interactions to attempt could help reduce this number drastically, e.g. by just focusing on right swipes and single taps.

RQ1. How comprehensively and efficiently can our technique capture and perform gesture interactions?

To answer RQ1 we conclude that our approach can successfully handle the majority of ways to implement gestures in the browser, though it might be necessary with some extra configuration to support specific libraries using custom event types. Additionally, to increase precision of state and transition discovery developers may want to minimize what gesture interactions are attempted when crawling a site.

B. Case Study (for RQ2 and RQ3)

We proceeded to test out this tool on 6 actual websites to evaluate the remaining two research questions. The target websites were chosen due to having advanced GUI's, being tailored for mobile, and being crawlable through the MITM proxy. C1 is an interactive application for manipulating animations, while C2 implements a MS Paint clone in the web browser. C3 is a web application for displaying presentations. C4 and C5 are informational sites customized for smartphone devices. We kept the level of state abstraction low, trying to consider all changes introduced to the DOM. However, in order to weed out false positives we do instruct mobCrawler to ignore ads and automatically created hash values on a site-by-site basis. Maximum crawl times are set to 2 hours, with a crawl depth of 2. We also instruct the browser to not accept any cookies.

TABLE III. CASE STUDY TARGET SITES

Case	URL	Mobile UA
C1	bomomo.com	No
C2	muro.deviantart.com	No
C3	slides.com/andreylisin/omaha-server#/6/1	Yes
C4	m.weather.com/weather/today/JAXX0085:1:JA	Yes
C5	touch.toyota.com/index.html	Yes

TABLE IV. CRAWL RESULTS

	C1	C2	C3	C4	C5
States, "click"	22	60	18	60	50
States, other	42	124	61	7	38
Transitions, "click"	22	60	23	61	100
Transitions, other	42	124	198	7	108
Transitions, <a> + <button>	4	16	0	29	49
Transitions, other elements	60	175	221	39	159

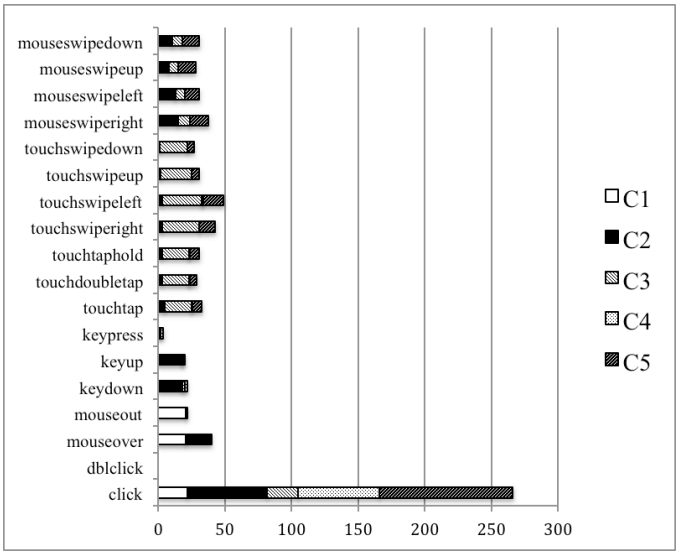

Figure 5. Event Type Distibution for State Transitions

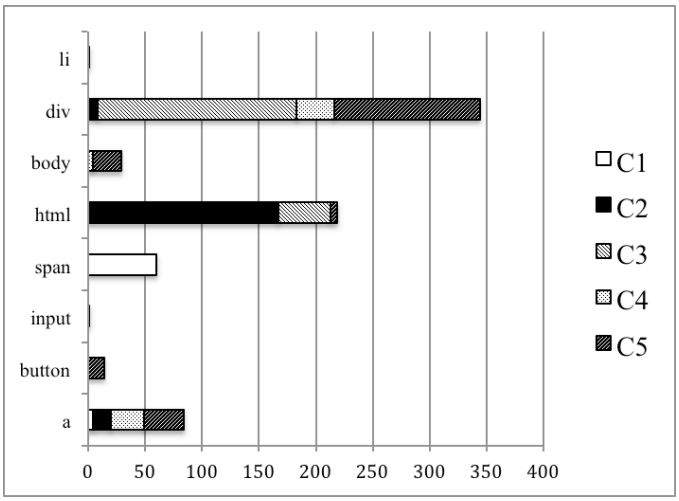

Figure 6. Element Type Distribution for State Transitions

As can be seen in Table IV, all the target sites have a large number of states and transitions being reached by exercising interactions other than "click". In C1 a lot of the menu buttons' CSS styling are changed as part of mouseover and mouseout events. C2 mirrors C1 in that most of the non-"click" induced states are often differentiated by changes in styling or visibility of elements. There are also a number of keyboard shortcuts to quickly select given menu options.

C3 depends on swipe gestures to switch between various slides. The high number of transitions can be explained by the fact that a lot of states can be reached by multiple interactions. For example, moving to the next slide in the deck can be accomplished by swiping right or using a button in the control panel. C4 and C5 are examples of more standard mobile web applications, and we can see that most states can be found using the "click" interaction. However, both sites contain image galleries or sections which can be swiped to change the contained content, similar to our motivating example. C5 has a huge number of transitions due to multiple interactions reaching the same state, as was also observed in C3.

Looking at Figure 5, we can see that "click" events still, unsurprisingly, still make up the major amount of transitions when crawling, though a considerable amount of transitions use other event types. In C3-C5 there are considerably many touch transitions, though many lead to the same state. Swiping horizontally seems to be more fruitful than vertically. Both the desktop sites, C1 and C2, have a fair amount of mouse and keyboard events. Though not inducing a state change we observed the swipes were also able to change what was drawn on the <canvas> element.

RQ2. How often do various keyboard, mouse, and touch events lead to new state transitions in modern RIA's, and which event types are more likely to induce new states?

To answer RQ2, we conclude that swipe gesture transitions are very common for mobile web applications and can induce new and meaningful states. The huge number of transitions leading to the same state might be unwanted if crawling with the aim of discovering content, but may be desirable if trying to generate a comprehensive test suite or detecting errors in a web application. For desktop sites, both keyboard and mouse events are found to be common for advanced web applications with advanced GUI's.

Considering the amount of non-"click" transitions in C2, C3, and C5, it seems they are too numerous to ignore. Though not all applications rely heavily on these kinds of transitions, such as C4, the number can be too high to simply ignore. As evidenced by the image galleries in C5 and slides in C3, content crawlers may be able to increase the amount of data found if they attempt to emulate swipe gestures on mobile websites. When it comes to automatically generating test suites for web applications, the developers may get a higher test coverage by instructing the crawler to attempt gesture types they know are present in the application.

Figure 6 depicts what elements were interacted with to find state transitions. As can be seen, this varies depending on the specific website, with C1 relying heavily on elements which it uses for its menu buttons. C2, C3 and C5 have large

numbers of event handlers attached to the <html> or <body> element. Many of these transitions are gestures which can be performed anywhere in the visible document. <a> and elements are in consistent use in all the web sites. However <a> and <button> elements are not nearly as prominent as the element in terms of inducing transitions.

RQ3. What DOM elements are more likely to be targets for interactions leading to new state transitions?

To answer RQ3 we can conclude that element types employed as interaction targets can differ widely depending on the website. <a> and elements seem to be the most common interaction targets to use for transitioning between states. Web applications using gesture actions applied to the entire documents surface area will likely have interaction targets on the <html> or <body> element. The results indicate that event handler analysis is a more comprehensive and precise way of finding possible transitions in a state than exhaustively trying out a predetermined list of element types in each state.

The most obvious takeaway is that the types of elements used for inducing transitions are too varied from site to site, to rely on a small subset when targeting a large variety of web applications. This could be useful when performing empirical studies on a large host of web applications by way of crawling. By combining event handler analysis along with a somewhat selective set of gesture interactions to try out, the resulting state graphs could be richer and more representative of their respective web applications.

V. USAGE

Our findings indicate our technique can help augment existing tools that are automatically exploring web applications by increasing the number of found transitions. If our case study is a good indicator of a more general trend in web applications, there are many web sites containing advanced interactions which can be modeled better than with current crawling techniques. A fitting example would be content crawling mobile web applications relying on swiping to load new content. If a crawler can't emulate a user swiping on the given GUI element, the content loaded as a result may be left unexplored and unindexed.

Another area in which our technique seemingly can help is increasing code coverage in automatically created test suites for web applications relying on advanced interactions. This is well illustrated by C3 in our case study, which is a web application for browsing slide presentations in a mobile browser. When creating an automatic test suite for this application, it is reasonable to assume the developer also wants to generate test cases for the various gestures involved in manipulating the slides. By emulating these gestures we could trigger event handlers not triggered by traditional crawling techniques and thus increase the total code coverage of the web application. Possibly this could also lead to finding a number of unexpected errors and event sequences, and should generally increase the quality of the test suite.

For many of the same reasons stated in the previous two examples, we believe our approach would be beneficial for various automatic evaluation techniques of rich web applications. Several approaches to accessibility and security evaluation rely on automatically exercising the GUI to find states and transitions as evidenced in [6] and [5] respectively. It seems reasonable to assume these approaches would benefit from our technique as they would be able to explore a larger number of transitions initiated by non-"click" interactions. Once again it seems like mobile web applications especially would benefit from this.

VI. RELATED WORK

Though Crawljax[1] can be configured to initiate simple click events on any kind of element, it relies on exhaustively exercising all such elements on a page rather than finding only the elements with "click" event handlers attached on them. The support for exercising different kinds of event types is minimal and focuses on simple mouse clicks. Our extensions allow it to find all desired events registered on a page via event handler analysis, and exercise simple even dispatches as well as single-point gestures.

There has been performed some previous research utilizing event handlers as evidenced by tools such as ARTEMIS from [3] and FEEDEX from [12]. These tools seem to analyze event handler registrations by using a modified version of the Rhino JavaScript interpreter. They use the event handlers as input in prioritizing what crawl action to take next. [18] introduces a symbolic execution framework for finding security vulnerabilities in web applications. It utilizes event handler analysis and can perform simple event dispatches, but does not attempt to emulate gestures. [15] introduces a static analysis approach to creating finite state machines from source code by analyzing event handlers. In contrast to out tool, none of these approaches try to find and emulate more complex gestures, and most of them, except [18], also seem to ignore simple event dispatches.

A static approach to analyzing interactions in JavaScript applications can be found in [23] which models web applications as finite state machines. However, it focuses on verifying interaction invariants by way of model checking as well as mutation analysis, while we are following a dynamic approach of modeling web applications as state diagrams for purposes such as test suite generation and content indexing. Related papers [22, 24] use the same approach, and target mutation analysis and testing AJAX and RIA's.

There has also been performed related research in the native mobile application space. [19] targets Android applications, and introduces a technique in which they first analyze event handlers, and then generate and exercise different sequences of events to reach hard-to-find states and transitions. [20] and [21] automatically analyzes event handlers in order to create and exercise events to find GUI bugs or create test suites for Android applications. While all these researches focus on apps on the Android platform, our research is focused on RIA's on the web.

VII. Conclusion

We have introduced a new technique for finding and exercising mouse, keyboard, and touch candidate interactions when crawling web applications, as well as a case study investigating its performance in modern RIA's. The technique we proposed seems like a good fit for automatic crawling of web applications with advanced interaction models, especially mobile. Automatic test generation and error detection, as well as automatic solutions for evaluating accessibility and usability seem like good use cases.

In the future we would like to replace the script injection mechanism with something less intrusive than a MITM proxy, as well as add a default configuration supporting the custom event types of all major gesture libraries. In addition, we believe it would be fruitful to do a bigger case study involving a number of the most popular mobile web applications to further investigate the applicability of the approach. This would give us a better idea of the robustness of the approach and whether the patterns observed in our limited case study holds on a larger scale.

References

[1] A. Mesbah, E. Bozdag, and A. van Deursen. "Crawling Ajax by inferring user interface state changes," Eighth International Conference on Web Engineering, 2008, pp. 122-134. IEEE.

[2] G. Richards, S. Lebresne, B. Burg, and J. Vitek. "An analysis of the dynamic behavior of JavaScript programs," Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation, 2010, pp. 1-12. ACM.

[3] S. Artzi, J. Dolby, S. H. Jensen, A. Moller, and F. Tip. "A framework for automated testing of javascript web applications," 33rd International Conference on Software Engineering, 2011, pp. 571-580. IEEE.

[4] D. Roest, A. Mesbah, and A. van Deursen. "Regression testing ajax applications: Coping with dynamism," Third International Conference on Software Testing, Verifiation and Validation, 2010, pp. 127-136. IEEE.

[5] C. P. Bezemer, A. Mesbah, and A. van Deursen. "Automated security testing of web widget interactions," Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, 2009, pp. 81-90. ACM.

[6] F. Ferrucci, F. Sarro, D. Ronca, and S. Abrahao. "A Crawljax Based Approach to Exploit Traditional Accessibility Evaluation Tools for AJAX Applications," Information Technology and Innovation Trends in Organizations, 2011, pp. 255-262. Springer.

[7] C. Duda, G. Frey, D. Kossmann, and C. Zhou. "Ajaxsearch: crawling, indexing and searching web 2.0 applications," Proceedings of the VLDB Endowment 1.2, 2008, pp. 1440-1443. ACM.

[8] comScore. "The U.S. Mobile App Report," comScore Whitepaper.

[9] StatCounter Web Usage Stats, Jan 2012 – Jan 2015. http://gs.statcounter.com/#all-comparison-ww-monthly-201201-201501. 12 May 2015.

[10] A. Nederlof, A. Mesbah, and A. van Deursen. "Software engineering for the web: the state of the practice," Companion Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 4-13. ACM.

[11] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra. "Guided test generation for web applications," 35th International Conference on Software Engineering, 2013, pp. 162-171. IEEE.

[12] A. M. Fard, and A. Mesbah. "Feedback-directed exploration of web applications to derive test models," ISSRE, 2013, pp. 278-287.

[13] A. Marchetto, P. Tonella, and F. Ricca. "State-based testing of ajax web applications," 1st International Conference on Software Testing, Verification, and Validation, 2008, pp. 121-130. IEEE.

[14] Mitmproxy. https://mitmproxy.org/. 12 May 2015.

[15] Y. Maezawa, H. Washizaki, and S. Honiden. "Extracting interaction-based stateful behavior in rich internet applications," 16th European Conference on Software Maintenance and Reengineering, 2012, pp. 423-428. IEEE.

[16] H. Takagi, S. Saito, K. Fukuda, and C. Asakawa. "Analysis of navigability of Web applications for improving blind usability." ACM Transactions on Computer-Human Interaction v. 14, no. 3, 2007, article number 13. ACM.

[17] W3 JavaScript APIs. http://www.w3.org/TR/#tr_Javascript_APIs/. 12 May 2015.

[18] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. "A symbolic execution framework for javascript," IEEE Symposium on Security and Privacy, 2010, pp. 513-528. IEEE.

[19] C. S. Jensen, M. R. Prasad, and A. Møller. "Automated testing with targeted event sequence generation," Proceedings of the 2013 International Symposium on Software Testing and Analysis, 2013, pp. 67-77. ACM.

[20] C. Hu, and I. Neamtiu. "Automating GUI testing for Android applications," Proceedings of the 6th International Workshop on Automation of Software Test, 2011, pp. 77-83. ACM.

[21] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. de Carmine, and A. M. Memon. "Using GUI ripping for automated testing of Android applications," Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, 2012, pp. 258-261. ACM.

[22] K. Nishiura, Y. Maezawa, H. Washizaki, S. Honiden, "Mutation Analysis for JavaScript Web Applications Testing," Proceedings of the 24th International Conference on Software Engineering and Knowledge Engineering, 2013, pp.159-165.

[23] Y. Maezawa, H. Washizaki, Y. Tanabe and S. Honiden, "Automated Verification of Pattern-based Interaction Invariants in Ajax Applications," IEEE/ACM 28th International Conference on Automated Software Engineering, 2013, pp. 158-168. IEEE.

[24] Y. Maezawa, K. Nishiura, H. Washizaki, S. Honiden, "Validating Ajax Applications Using a Delay-Based Mutation Technique," Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014, pp. 491-502. ACM.