jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications

Giancarlo Pellegrino[1](✉), Constantin Tschürtz[2], Eric Bodden[2], and Christian Rossow[1]

[1] Center for IT-Security, Privacy, and Accountability (CISPA), Saarland University, Saarbrücken, Germany
{gpellegrino,crossow}@cispa.saarland
[2] Secure Software Engineering Group, Technische Universität Darmstadt, Darmstadt, Germany
constantin.tschuertz@gmail.com, eric.bodden@sit.fraunhofer.de

Abstract. Web application scanners are popular tools to perform black box testing and are widely used to discover bugs in websites. For them to work effectively, they either rely on a set of URLs that they can test, or use their own implementation of a *crawler* that discovers new parts of a web application. Traditional crawlers would extract new URLs by parsing HTML documents and applying static regular expressions. While this approach can extract URLs in classic web applications, it fails to explore large parts of modern JavaScript-based applications.

In this paper, we present a novel technique to explore web applications based on the dynamic analysis of the client-side JavaScript program. We use dynamic analysis to hook JavaScript APIs, which enables us to detect the registration of events, the use of network communication APIs, and dynamically-generated URLs or user forms. We then propose to use a navigation graph to perform further crawling. Based on this new crawling technique, we present jÄk, a web application scanner. We compare jÄk against four existing web-application scanners on 13 web applications. The experiments show that our approach can explore a surface of the web applications that is 86 % larger than with existing approaches.

1 Introduction

Web application scanners are black box security testing tools that are widely used to detect software vulnerabilities in web applications. As a very essential component, the scanners have to explore all parts of the web application under test. Missing functionality during this exploration step results in parts of the web application remaining untested—leading to potential misses of critical vulnerabilities. To addres this problems, scanners typically expand their initial set of seed URLs. That is, they *crawl* a web application to extract as many different URLs as possible. URLs are then used to send crafted inputs to the web application to detect vulnerabilities. Nowadays, crawlers find new URLs by pattern matching on the HTML content of web sites, e.g., using regular expressions. While this approach can extract URLs in classic web applications, it fails to explore large parts of modern web applications.

© Springer International Publishing Switzerland 2015
H. Bos et al. (Eds.): RAID 2015, LNCS 9404, pp. 295–316, 2015.
DOI: 10.1007/978-3-319-26362-5_14

The advent of JavaScript and client-side communication APIs has increased the complexity of the client-side of web applications. While in the past the client side was merely a collection of static HTML resources, in modern web applications the client side is a full-fledged program written in JavaScript running in a web browser. In these programs, URLs and forms are no longer only static objects, but they may also be the result of client-side computations. For example, JavaScript functions can be used to generate user login forms, to encode user inputs using non-standard HTML form encoding (e.g., JSON), and to include form input values at runtime. Prior work has shown that many URLs in modern web applications are generated dynamically by JavaScript code [1]. As web scanners tend to perform checks on the HTML code, they will fail to cover large parts of web applications. As a result, this leaves a significant fraction of the attack surface of a web application unknown to the underlying vulnerability testing methodology, resulting in incomplete tests.

However, crawling modern web applications is challenging. The difficulties mainly originate from new features introduced by JavaScript. JavaScript programs use an event-driven paradigm, in which program functions are executed upon events. To trigger the execution of these functions, and thus the generation of URLs, a web crawler needs to interact with the JavaScript program. Recently, Mesbah et al. have proposed to combine web-application crawling with dynamic program analysis to infer the state changes of the user interface [2]. However, this approach relies on a number of heuristics which do not cover all the interaction points of the client side. As a result, the largest part of the web application remains unexplored, which ultimately limits the capability to detect vulnerabilities.

In this paper, we address the shortcomings in terms of poor code coverage of existing crawling techniques. We propose a novel approach that combines classic web application crawling and dynamic program analysis. To this end, we dynamically analyze the web applications by hooking JavaScript API function and performing runtime DOM analysis. Using a prototype implementation called `jÄk`, we show that our methodology outperforms existing web application scanners, especially when it comes to JavaScript-based web applications. Whereas existing tools find only up to 44 % of the URLs, we show that `jÄk` doubles the coverage of the WIVET web application [3]. We also tested `jÄk` against 13 popular web applications, showing that in eleven cases it has the highest coverage as compared to existing tools. In summary, we make the following contributions:

- We present a novel dynamic program analysis technique based on JavaScript API function hooking and runtime DOM analysis;
- We propose a model-based web-application crawling technique which can infer a navigation graph by interacting with the JavaScript program;
- We implement these ideas in `jÄk`, a new open-source web application scanner. We compare `jÄk` against four existing scanners and show their limitations when crawling JavaScript client-side programs;
- We assess `jÄk` and existing tools on 13 case studies. Our results show that `jÄk` improves the coverage of web application by about 86 %.

2 Background

Before turning to our technique, we will briefly describe two JavaScript concepts that are often used in modern web applications. These two, events and modern communication APIs, severely increase the complexity of scans.

2.1 Event Handling Registration

Client-side JavaScript programs use an event-driven programming paradigm in which (i) browsers generate events when something interesting happens and (ii) the JavaScript program registers functions to handle these events. JavaScript supports different event categories: device input events (e.g., mouse move), user interface events (e.g., focus events), state change events (e.g., `onPageLoad`), API-specific events (e.g., Ajax response received), and timing events (e.g., timeouts). Event handlers can be registered via (i) event handler attributes, (ii) event handler property, (iii) the `addEventListener` function, or (iv) timing events:

Event Handler Attribute — The registration of an event handler can be done directly in the HTML code of the web application. For example, when the user clicks on the HTML link, the browser executes the code in the attribute `onclick`:

```
1 <a href="contact.php" onclick="doSomething"></a>
```

Event Handler Property — Similarly, event handlers can be registered by setting the property of an HTML element. Below is an equivalent example of the previous one. The code first defines a JavaScript function called `handler`. Then, it searches for the HTML element with the identifier `link`. Then, it sets the property `onclick` with the function `handler`. After that, whenever the user clicks on the link to `contact.php`, the browser executes the `handler` function.

```
1 <a id="link" href="contact.php"></a>
2
3 <script type="text/javascript">
4   function handler() { /* do something */ }
5   var link = document.getElementsById("link");
6   link.onclick = handler;
7 </script>
```

`addEventListener` Function — Third, programmers can use `addEvent Listener` to register events, as shown below. Again, this code searches the HTML element with ID `link`. Then, it calls `addEventListener()` with two parameters. The first parameter is the name of the event, in our case the string `"click"` (for the user click event). The second parameter is the name of the function, i.e., `handler`.

```
1 <a id="link" href="contact.php"></a>
2
3 <script type="text/javascript">
4   function handler() { /* do something */ }
5   var link = document.getElementsById("link");
6   link.addEventListener("click", handler);
7 </script>
```

Timing Events — Finally, timing events are fired only once after a specified amount of time, i.e., timeout event, or at regular time intervals, i.e., interval event. The handler registration for these events is performed via the `setTimeout` and the `setInterval` functions, respectively.

Modern web applications rely heavily on these events to trigger new behavior. Web application scanners thus have to support event-based code.

2.2 Network Communication APIs

The communication between the web browser and the server side has shifted from synchronous and message-based, to asynchronous and stream-oriented. Understanding and supporting modern network communication APIs is thus essential for web application scanners. For example, consider Listing 1.1, which shows the use of the XMLHttpRequest (XHR) API, in which the JavaScript program sends an asynchronous HTTP POST request to the server side.

Listing 1.1. XMLHttpRequest API Example

```
1  var server = "http://foo.com/";
2  var token = "D3EA0F8FA2"
3  var xhr = new XMLHttpRequest();
4  xhr.open("POST", server);
5  xhr.addEventListener("load", function() {
6    // process HTTP response
7  });
8  xhr.send("token=" + token);
```

The JavaScript program first initializes two variables: a URL that identifies the endpoint to which the HTTP request is sent, and a `token` that can be an anti-CSRF token or an API key to allow the client-side JavaScript program to access third-party web service. Then, the JavaScript program instantiates an XMLHttpRequest object for an HTTP POST request and registers a handler to process the server response. Finally, it sets the POST body as `token=D3EA0F8FA2`, and sends the HTTP request to the server.

Classic crawlers statically analyze the HTML and JavaScript code to extract URLs. This makes it hard for them to extract the correct endpoint. Furthermore, classic crawlers cannot extract the structure of the HTTP POST request. We find that four popular crawlers (w3af, skipfish, wget, and crawljax) cannot extract the POST request structure of this example. Two of these crawlers, w3af and skipfish, use regular expressions to extract strings that look like URLs, as a result they may find out URLs when stored in variables such as `server`, however, they will miss the POST parameter `key`. Worse, if the URL would have been generated dynamically, e.g., "`server="http://"+domain+"/";`", then w3af and skipfish could not detect even the first part. Finally, the two other crawlers, wget and crawljax, even fail to detect URLs stored in JavaScript variables. Many parts of modern web applications can only be reached by interpreting such dynamically generated requests, thus limiting the coverage of existing crawlers (cf. Sect. 5).

3 Crawling Modern Web Applications

As explained in the previous section, modern web applications can use JavaScript events to dynamically react to events, and to update the internal and visual state of the web application in response. Figure 1 gives a graphical representation of the page flow of an example toy web application. Initially, the user loads the URL http://foo.com/, which loads the web application's landing page into the browser. This page is then loaded into its initial state and displayed to the user. The user can then interact with the page, for instance submit HTML forms or click HTML links, which will invoke further pages such as http://foo.com/bar/, shown to the right. User events or spontaneous events such as timers can also, however, change the page's internal and visual state, as denoted by the dotted arrows. Those internal states can inflict significant changes to the page's DOM, which is why they should be considered by crawlers as well. Most current crawlers, however, will focus on HTML only, which restricts them virtually to discovering only those HTML page's initial states.

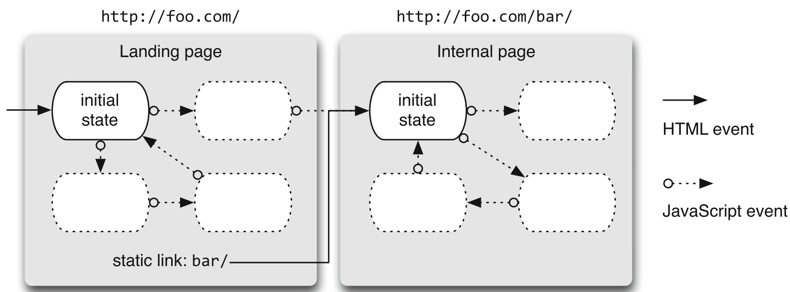

Fig. 1. State changes and page flow induced by clicks and events

We propose a new concept based on dynamic analysis for crawling web applications that overcomes the limitations of existing crawlers. The overall idea is to combine classic web application crawling with program analysis of the client-side of a web application. The crawler starts from a seed URL, e.g., the landing page, and it retrieves the resources of the client-side program, e.g., an HTML page or JavaScript program. Then, it runs the client-side program in a modified JavaScript execution environment to analyze its behavior. From the analysis, the crawler can extract events and URLs which are later used to explore both the client-side program and the server side. Finally, the crawler repeats the analysis until when no more new behaviors can be discovered. Section 3.1 presents our dynamic JavaScript program analyses. Section 3.2 presents the logic to expand the search via crawling.

3.1 Dynamic JavaScript Program Analysis

We deploy dynamic program analysis to monitor the behavior of the JavaScript program and extract events, dynamically-generated URLs and forms, and endpoints for the communication with the server side.

Dynamic analysis of client-side JavaScript programs can be performed in different ways. One approach is to modify the JavaScript interpreter to inspect and monitor the execution of the program. In this setting, whenever an instruction of interest executes, the interpreter executes a hook function instead of or in addition to the original instruction. However, this approach requires one to modify a JavaScript engine, most of which are notoriously complex pieces of software. Furthermore, this approach will bind the technique to a specific engine. Another way to perform dynamic analysis is to insert calls to own JavaScript functions within the source code of the client-side JavaScript program. This approach requires one to process and transform the source code of the program. Unfortunately, the source code of JavaScript programs may not be available as a whole as it may be streamed to the client side at run-time and one piece at a time.

`jÄk` follows a third option, namely monitoring the execution of the program by hooking functions to APIs inside the JavaScript execution environment. `jÄk` first initializes the JavaScript engine. Then it modifies the execution environment by running own JavaScript code within the engine. This code installs function hooks to capture calls to JavaScript API functions and object methods, and schedules the inspection of the DOM tree. After that, it runs the client-side JavaScript program.

In the remainder of this section, we detail these techniques. First, we present the basic techniques for performing function hooking in JavaScript. Then we describe the use of function hooking to capture the registration of event handlers and the use of network communication APIs, respectively. Finally we describe how dynamic traces are collected.

Function Hooking. Function hooking is a family of techniques that allows one to intercept function calls to inspect the parameters or alter the behavior of the program. In this section, we present two features of JavaScript that we use to hook functions: *function redefinition* and *set functions*.

Function redefinition is a technique for overwriting JavaScript functions and object methods. Consider the example in Listing 1.2, which shows the use of function redefinition that logs any call to the function `alert`. This is achieved first by associating a new name to the function `alert` (Line 2), and then by redefining the `alert` function (Line 3). The redefinition still behaves as the original `alert`, however, it adds (i.e., hooks) a call to log its use.

While function redefinition can be used to hook arbitrary functions to function calls, it cannot be used when functions are set as an object property, i.e., `obj.prop=function(){[...]}`. To hook functions in these cases, we use so-called *set functions*, which are bound to object properties that are called whenever the property is changed. For example, one can hook the function `myHook` to the property `propr` of the object `obj` as follows:

Listing 1.2. Function hooking via function redefinition

```
1 alert("Hello world!"); // show a popup window
2 var orig_alert = alert;
3 function alert(s) {
4   console.log("call to alert" + s); // hook
5   return orig_alert(s);
6 }
7 alert("Hello world!"); // message is also shown in the console
```

`Object.defineProperty(obj,"prop",{set:myHook})`.

Event Handlers Registration. We now show the use of function hooking to capture the registration of event handlers in three of the four registration models: `addEventListener` function, event handler property, and timing events. For the fourth registration model, i.e., event handler attribute, we do not use function hooking. As in this model handlers are registered as HTML attribute, we captured them by visiting the HTML DOM tree.

Hooking `addEventListener` — To capture the registration of a new handler, `jÄk` injects its own function whenever the `addEventListener` function is called. Listing 1.3 shows an example for the hooking code. The function `installHook` installs a hook function `hook` before the execution of a function `f` of object `obj`. `installHook` first preserves a reference to the original function (Line 2). Then, `jÄk` replaces the original function with its own anonymous function (Line 3 to Line 6). The anonymous function first calls the hook (Line 4) and then the original function (Line 5). Here, the parameters of `hook` are `this` and `arguments`. Both parameters are JavaScript keywords. The first one is a reference to the object instance whereas the latter is a list containing the parameters that will be passed to the function `f`. Finally, `jÄk` can use the `installHook` function to install its hook handler `myHook` for every call to the function `addEventListener` of *any* HTML tag element, as shown below:

```
1 installHook(Element.prototype, "addEventListener", myHook)
```

Here `Element.prototype` is a special object that defines the basic behaviors of all DOM nodes.

Hooking Event Handler Properties — To capture the registration of event handlers via event properties, one can install a hook function as a set function in the

Listing 1.3. Function Hooking for the addEventListener function

```
1 function installHook(obj, f, hook) {
2   var orig = obj[f];
3   obj[f] = function() {
4     hook(this, arguments);
5     return orig.apply(this, arguments);
6   }
7 }
```

DOM elements. However, this approach requires further care. First, the registration of a set function may be overwritten by other set functions installed by the JavaScript program. As opposed to function redefinition, set functions do not guarantee that the hook will remain for the entire duration of the analysis. This can be solved by first redefining the `defineProperty` function and then monitoring its use. Then, if the hook detects a set-function registration, it will create a new set function which chains `jÄk`'s set function with the one provided by the program. Second, we observed that the registration of set functions for event properties may prevent the JavaScript engine from firing events. This can interfere with the operations of a JavaScript program, e.g., it can break the execution of self-submitting forms[1]. While `jÄk`'s set function will still detect event handler registrations, after the discovery, the JavaScript engine needs to be reinitialized.

Finally, as opposed to function redefinitions, this technique may not work in every JavaScript engine. To install set functions, the JavaScript engine needs to mark the property as *configurable*. Unfortunately, this is an engine-dependent feature. For example, the JavaScript engines of Google and Mozilla, i.e., V8 and SpiderMonkey, support this feature whereas the JavaScript engine of Apple does not allow it. When function hooking on event properties is not allowed, one can instead inspect the DOM tree to detect changes in the event properties.

Hooking Timing Event Handlers — To capture the registration of timing event handlers, it is possible to reuse the `installHook` function of Listing 1.3 as follows:

```
1 installHook(window, "setTimeout", myHook)
2 installHook(window, "setInterval", myHook)
```

where `myHook` is the hook function.

Network Communication APIs. Next, we describe the use of function hooking to dynamically inspect the use of networking communication APIs. We will illustrate an example hooking the XMLHttpRequest API, but the general approach can easily be extended to further communication APIs.

As shown in Sect. 2.2, the XHR API is used in three steps. First, an XHR object is instantiated. Then, the HTTP request method and the URL of the server side is passed to the XHR object via the `open` function. Finally, the HTTP request is sent with the function `send`. `jÄk` can use `installHook` to inject its hook handler `myHook` for both `open` and `send` as follows:

```
1 installHook(XMLHttpRequest, "open", myHook);
2 installHook(XMLHttpRequest, "send", myHook);
```

Other network communication APIs may require the URL of the endpoint as a parameter to the constructor. For example, WebSocket accepts the URL only in the constructor function as follows: `var ws = new WebSocket(server)`. In general, when one would like to hook a function in the constructor, Line 5 of `installHook` in Listing 1.3 needs to be modified to return an instance of the object, i.e., `return new orig(arguments[0], ...)`.

[1] A self-submitting form is an HTML form that is submitted by firing submit or mouse click events within the JavaScript program.

Run-Time DOM Analysis. The DOM tree is a collection of objects each representing an element of the HTML document. The DOM tree can be visited to inspect its current state. Each visit can be scheduled via JavaScript events or it can be executed on-demand. In this paper, we consider three uses of run-time DOM analysis. First, it is used to extract the registration of handlers as HTML attributes. Second, it is used to identify changes in the tree while firing events. Third, it can be used to discover the registration of event handlers when the JavaScript engine does not allow to hook code as set functions.

Collection of Dynamic Traces. After describing how to install jÄk's hook functions, we now turn to the actual behavior of the these functions. In general, jÄk uses hook functions to collect information from the run-time environment at the point of their invocation. This information is then sent to the crawler, which collects them in an execution trace.

For the event handler registration, the hook function depends on the type of event (see, e.g., Listing 1.4). For example, for DOM events, the hook function collects the name of the event, the position in the DOM tree of the source and sends it to the crawler. Instead, for timing events, the hook can collect the timeout set by the caller. In either case, hook functions send trace entries to the crawler via a JavaScript object `trace`, which is mapped to a queue object in the crawler's memory. This object acts as a bridge between the JavaScript execution environment and the crawler's execution environment.

Listing 1.4. Hook Function for the addEventListener and setTimeout

```
1  function addEventListenerHook(elem, args) {
2    path = getPath(elem);
3    entry = {
4      "evt_type"   : args[0], //1st par of addEventListener
5      "evt_source" : path
6    };
7    trace.push(entry);
8  }
```

```
1  function timeoutHook(elem, args) {
2    entry = {
3      "evt_type"   : "timeout",
4      "time"       : args[1] //1st par of setTimeout
5    };
6    trace.push(entry);
7  }
```

When collecting trace entries for network communication APIs, one has to address two issues. First, the APIs typically require multiple steps to set up a communication channel and to deliver messages to the server side. For example, the XHR API requires at least three steps (Lines 3–8 in Listing 1.1). These steps are not necessarily atomic. In fact, a program may open a pool of XHR connections, and finally call the `send` function of each object. In this case, a single hook will result in a trace which contains uncorrelated trace events: at the beginning a sequence of "open" events with the URL endpoints, and then a sequence of only "send" events with the body being sent.

For these reasons, jÄk defines a hook for each of the API functions and then uses the API object to store the current state of the API calls. For example, Listing 1.5 shows the hook function for the API functions `open` and `send`. The hook function `xhrOpenHook` creates two new object properties in the object `xhr` for the HTTP request method and the URL, respectively. Then, the function `xhrSendHook` collects the content of the two object properties and the body of the HTTP requests, and the sends them to the crawler. Such hooks are thread-safe, and thus even work correctly when JavaScript programs access the network communication API concurrently (e.g., within Web Workers [4]).

3.2 Crawling

In the previous section, we presented the dynamic analysis technique in isolation. In this section, we will integrate the dynamic analysis into our web crawler jÄk. The crawler is model-based, i.e., it creates and maintains a model of the web application which is used at each step to decide the next part to explore. First, we describe how we create the model. Then, we discuss the selection criteria for the next action, and finally the termination conditions.

Navigation Graph. jÄk creates and maintains a navigation graph of the web application which models both the transitions within a client-side program and the transitions between web pages. The model is a directed graph similar to the one shown in Fig. 1, in which nodes are clusters of pages and edges can be events and URLs. Each page p is modeled as a tuple of three elements $p = \langle u, E, L, F \rangle$ where u the web page URL, E is the JavaScript events, L a set of URLs (e.g., linked URLs, server-side endpoints), and F a set of HTML forms. jÄk normalizes all URLs by striping out query string values and sorting the query string parameter lexicographically. Two pages p' and p'' are in the same cluster if (i) u' and u'' are identical and (ii) the two pages are *sufficiently similar*. The similarity is calculated as a proportion between the number of common events, URLs and forms over the total number of events, URLs and forms. Precisely, the similarity is defined as follow:

Listing 1.5. Hook Functions for XHR API

```
1  function xhrOpenHook(xhr, args) {
2    xhr.method = args[0]; //1st par of HTMLHttpRequest.open, i.e., HTTP
         method
3    xhr.url = args[1];    //2nd par, i.e., the URL
4  }
5  function xhrSendHook(xhr, args) {
6    entry = {
7     "evt_type"   : "xhr",
8      "url"        : xhr.url,
9      "method"     : xhr.method,
10     "body"       : args[0] //1st par of XMLHttpRequest.send
11   };
12   trace.push(entry);
13 }
```

$$s(p', p'') = \frac{|E' \cap E''| + |L' \cap L''| + |F' \cap F''|}{|E' \cup E''| + |L' \cup L''| + |F' \cup F''|}$$

Through experimental analysis we determined that a similarity threshold of 0.8 generates the best results for our setting.

Navigating. The dynamic analysis of the JavaScript program generates a run-time trace containing event handler registrations and dynamically-generated URLs. It also includes the result of the DOM-tree analysis such as linked URLs and forms. This information is then sorted into two lists, a list of events and a list of URLs. These lists represent the *frontier of actions* that the crawler can take to further explore the web application.

Each type of action may have a different result. On the one hand, the request of a new URL certainly causes to retrieve a new page and, if the page contains a JavaScript program, then it is executed in a new JavaScript environment. This is not necessarily the case of events. Firing an event may allow the crawler to explore more behaviors of the JavaScript program, i.e., to generate new URLs. However, events may also cause to run a new JavaScript program, for instance by setting `window.location` to a new URL. However, we can block this behavior via function hooking. For these reasons, our crawler gives a higher priority to events with respect to the URLs. When no more events are left in the list, then we process the list of URLs. When all the lists are empty, then the crawler exits.

Visiting the Client-side Program — Events such as click, focus, double click, and mouse movements can be fired within the JavaScript execution environment. To fire an event e, `jÄk` first identifies the DOM element and then fires the event via the DOM Event Interface [5] function `dispatchEvent`. After that, `jÄk` observes the result of the execution of the handler via the dynamic analysis. The event handler can cause a refresh of the page, a new page to be loaded, a message to be sent to the server side. To avoid any interference with the server side, when firing events, the hook functions, e.g., for network communication API, will block the delivery of the message.

After having fired an event, `jÄk` can distinguish the following cases. If the event handler results into a network communication API, then `jÄk` takes the URL from the trace, and enqueues it in the list of URLs. Similarly, if the event handler sets a new URL (i.e., `window.location=`URL), then `jÄk` enqueues the URL into the linked-URLs list. If the event handler adds new linked URL and forms, then they are inserted into the appropriate list. Finally, if the event handler registers new events, then `jÄk` prepares the special event which comprises the sequence of events that lead to this point, e.g., $\hat{e} = \langle e, e' \rangle$ where e is the last fired event and e' is the newly discovered event. Then, $\hat{e}$ is added to the list of events. When the crawler schedules this event to be fired, it fires the events in the given order, i.e., first e and then e'.

Requesting New Pages — The crawler should aim to find pages that contain new content rather than pages with known content. To select the next page, jÄk assigns a priority to each of the URLs in the frontier based on two factors: (i) how many times jÄk has seen a similar URL in the past, and (ii) how scattered over the clusters past URLs are. The priority is thus calculated as the number of similar past URLs over the number of clusters in which the past URLs have been inserted in. If a URL in the frontier was never seen in the past, i.e., the priority is 0, then we force its priority to 2. The crawler processes URLs from the highest to the lower priority.

Termination. Without any further control on the behavior of the crawler, the crawler may enter a loop and never terminate its execution. jÄk thus uses two techniques to terminate its execution. First, it has a hard limit for the search depth. Second, the crawler terminates if it cannot find new content anymore. We describe the termination criteria in the following.

Hard Limits — Crawlers can enter loops in two situations. First, loops can happen across the different web pages of a web applications. This can be caused when crawling infinite web applications such as calendars or, for example, when two pages link to each other. The crawler may visit the first page, then schedule a visit to the second, which again points to the first page. These problems can be solved with a limit on the maximum search depth of the crawler. When the crawler reaches a limit on the number of URLs, it terminates the execution. Second, loops may also occur within single web pages. For example, the handler of an event can insert a new HTML element into the DOM tree and register the same handler to the new element. Similarly as seen for URLs, one can limit the maximum depth of events that can be explored within a web page. When the limit is reached, the crawler will no longer fire events on the same page.

Convergence-based Termination — In addition to these limits, the crawler terminates when the discovered pages do not bring any new content. The notion of *new* content is defined in terms of number of similar pages that the crawler visited in the past. To achieve this, the crawler uses the navigation graph and a limit on the number of pages per cluster. If the cluster has reached this limit, the crawler marks the cluster as *full* and any subsequent page is discarded.

4 Implementation of jÄk

This section presents our actual implementation of jÄk, our web-application scanner which implements the crawler and the program analysis presented in Sect. 3. jÄk is written in Python [6] and based on WebKit browser engine [7] via the Qt Application Framework bindings [8]. We released jÄk at https://github.com/ConstantinT/jAEk/.

jÄk comprises four modules: *dynamic analysis* module, *crawler* module, *attacker* module, and *analysis* module. The *dynamic analysis* module implements the techniques presented in Sect. 3.1. jÄk relies on the default JavaScript

engine of WebKit, i.e., the JavaScriptCore, to perform the dynamic analysis. Unfortunately, JavaScriptCore sets the event properties as not configurable. As a result, JavaScriptCore does not allow to use function hooking via set functions. To solve this, jÄk handles these cases via DOM inspection. However, we verified that the JavaScript engines of Google and Mozilla, i.e., V8 [9] and SpiderMonkey [10], allow one to hook set functions. In the future, we plan to replace the JavaScriptCore engine with V8.

The *crawler* module implements the crawling logic of Sect. 3.2. Starting from a seed URL, jÄk retrieves the client-side program and passes it to the dynamic analysis module. The dynamic analysis module returns traces which are used to populate the frontiers of URLs and events. Then, jÄk selects the next action and provides it to the dynamic analysis module. Throughout this process, jÄk creates and maintains the navigation graph of the web application which is used to select the next action. The output of the crawler module is a list of forms and URLs.

Finally, the *attacker* and *analysis* modules test the server side against a number of vulnerabilities. For each URL, the attacker module prepares URLs carrying the attack payload. Then, it passes the URL to the dynamic analysis module to request the URL. The response is then executed within the dynamic analysis module, which returns an execution trace. The analysis module then analyzes the trace to decide if the test succeeded.

5 Evaluation

We evaluate the effectiveness of jÄk in a comparative analysis including four existing web crawlers. Our evaluation consists of two parts. Section 5.1 assesses the capability of the crawlers based on the standard WIVET web application, highlighting the need to integrate dynamic analysis to crawlers. Then, in Sect. 5.2, we evaluate jÄk and the other crawlers against 13 popular web applications.

For our experiments, we selected five web crawlers: Skipfish 3.10b [11], W3af 1.6.46 [12], Wget 1.6.13 [13], State-aware crawler [14], and Crawljax 3.5.1 [2]. We selected Skipfish, W3af, and Wget as they were already used in a comparative analysis against State-aware crawler by prior work (see Doupé et al. [14]). Then, we added Crawljax as it is a crawler closest to our approach.

In our experiment, we used the default configuration of these tools. When needed, we configured them to submit user credentials or session cookies. In addition, we configured the tools to crawl a web application to a maximum depth of four. Among our tools, only W3af does not support bounded crawling[2].

[2] W3af implements a mechanism to terminate which is based on the following two conditions. First, W3af does not crawl twice the same URL and then it does not crawl "similar" URLs more than five times. Two URLs are similar if they differ only from the content of URL parameters.

5.1 Assessing the Crawlers' Limitations

First, we use the Web Input Vector Extractor Teaser (WIVET) web application [3] to assess the capabilities of existing crawlers and compare these to jÄk. The WIVET web application is a collection of tests to measure the capability of crawlers to extract URLs from client-side programs. In each test, WIVET places unique URLs in a different part of the client-side program including in the HTML and via JavaScript functions. Then, it waits for the crawler to request the URLs. WIVET tests can be distinguished in static and dynamic tests. A test is static if the unique URL is placed in the HTML document without the use of a client-side script. Otherwise, if the client-side program generates, requests, or uses URLs, then the test is dynamic. WIVET features 11 static tests and 45 dynamic tests. We focus on the dynamic behavior of client-side programs and thus limit the evaluation to running the 45 dynamic tests.

Table 1. Number and fraction of dynamic test passed by the different crawlers

Dynamic test categories	Total	Crawljax	W3af	Wget	Skipfish	jÄk
C1 Adobe Flash event	**2**	0	2	0	0	0
C2 URL in tag	**5**	5	5	0	5	4
C3 JS in URL, new loc.	**2**	2	1	0	1	2
C4 URL in tag, tim. evt.	**1**	0	1	0	1	1
C5 Form subm., UI evt.	**2**	2	1	0	1	1
C6 New loc., UI evt.	**27**	0	6	0	6	26
C7 URL in tag, UI evt.	**2**	0	2	0	1	2
C8 XHR	**4**	0	2	0	2	4
Total	**45**	**9**	**20**	**0**	**17**	**40**
In %	**100**	**20**	**44**	**0**	**38**	**89**

As URLs can be placed and used by the JavaScript program in different ways, we manually reviewed WIVET's dynamic tests and grouped them into eight classes. We created these classes by enumerating the technique used by each test. For example, we considered whether a test dynamically places an URL in an HTML tag, if the URL is for Ajax requests, or whether the action is in an event handler. Table 1 shows the eight classes and details the results of each crawler for each class.

As Table 1 shows, all tested crawlers but jÄk fail in more than half of the tests. In average, these tools passed only 25 % of the tests. With the exception of Wget, which failed all the dynamic tests, the success rate ranges from 20 % of Crawljax to 44 % of W3af. jÄk instead passed 89 % of the tests. For the event-based tests (i.e., **C4-7**), W3af, Skipfish and Crawljax succeeded in about 16 % of the tests, whereas for the server communication API tests (i.e., **C8**) they succeeded in 25 % of the tests. By comparison, jÄk achieved 96 % and 100 % of success rate for the classes **C4-7** and **C8**, respectively.

We next discuss the details of these experiments per tool. In total, jÄk passed 40 dynamic tests (89 %). With reference to the classes **C4-7**, jÄk discovered the

registration of the events via the hook functions. Then, it fired the events which resulted in the submission of the URL. In only one case, jÄk could not extract the URL which is contained in an unattached JavaScript function. As jÄk uses dynamic analysis, it cannot analyze code that is not executed, and thus it will not discover URLs in unattached functions. Nevertheless, jÄk could easily be extended with pattern-matching rules to capture these URLs. In fact, Skipfish and W3af were the only ones able to discover these URLs. For the class **C8**, jÄk discovered the URL of the endpoint via the hook functions and via DOM tree inspection. In this case, the test requested the URL via the XHR API and inserted it in the DOM tree.

jÄk failed in other four dynamic tests. First, one test of **C2** places a JavaScript statement `javascript:` as action form. jÄk correctly extracts the URLs, however it does not submit to the server side because jÄk does not submit forms during the crawling phase. Other two tests are in the class **C1**. This class contains tests which test the support of ShockWave Flash (SWF) objects. This feature is not currently supported by jÄk. Then, the last test is in **C5**. This test submits user data via a click event handler. jÄk correctly detects the event registration and it fires the event on the container of the input elements. However, the handler expects that the event is fired over the submit button element instead of the container. This causes the handler to access a variable with an unexpected value. As a result, the handler raises an exception and the execution is halted. In a web browser, the execution of the handler would have succeeded as a result of the propagation of the click event from the button to the outer element, i.e., the container[3]. The current version of jÄk does not support the propagation of events, instead it fires an event on the element where the handler has been detected, in this case the container.

Crawljax succeeded only in 9 out 45 tests (20 %). Most of the failed tests store URLs either in the location property of the window object (i.e., classes **C4** and **C6**), or as URL of a linked resource (i.e., class **C7**). The URL is created and inserted in the DOM tree upon firing an event. While Crawljax can fire events, it supports only a limited set of target HTML tags to fire events, i.e., buttons and links. Finally, Crawljax failed in all of the dynamic tests involving Ajax requests (See **C8**).

Skipfish and W3af performed better than Crawljax, passing 38 % and 44 % of the dynamic tests, respectively. These tools extract URLs via HTML document parsing and pattern matching via regular expression. When a URL cannot be extracted from the HTML document, the tools use pattern recognition via regular expressions. This technique may work well when URLs maintain distinguishable characteristics such as the URL scheme, e.g., `http://`, or the URL path separator, i.e., the character “`/`”. However, this approach is not sufficiently generic and cannot extract URLs that are composed dynamically via string concatenation. This is the case for the class **C6** Table 1 in which W3af and Skipfish passed only six tests out of 27. In these six tests, the URL is placed

[3] This model is the event *bubbling* and is the default model. Another model is the event *capturing* in which the event are propagated from the outermost to the innermost.

in a JavaScript variable and it maintains the URL path separator. With the use of regular expressions, W3af and Skipfish recognized the string as URL and submitted the server side thus passing the tests. However, in the remaining 21, URLs are created as the concatenation string variables and a JavaScript arrays. While regular expressions may be extended to include these specific cases, they will likely never be as complete as dynamic analysis.

5.2 Assessment Using Web Applications

Finally, we compare `jÄk` to the other crawlers by crawling popular web applications.

Table 2. Number of unique event-handler registrations extracted by `jÄk`, grouped by event category

Web Apps.	DDI	DII	UI	Chg	API	Errs.	Cust.	Total
WP	34	220	156	14	0	0	0	424
Gallery	930	7	1,257	23	0	0	303	2,520
phpBB	636	8	729	0	0	0	0	1,373
Joomla	46	144	232	26	0	0	0	448
Tidios w/ WP	14,041	26	3,715	192	111	12	641	18,738
Nibbleblog	12	42	0	7	0	0	0	61
Owncloud 8	826	905	274	53	44	0	134	2,236
Owncloud 4	126	651	234	68	10	0	36	1,125
Piwigo	1,609	1,323	281	44	0	0	40	3,297
Mediawiki	13,538	24,837	18,102	2,174	791	0	5,204	64,646
ModX	6,772	14,626	4,483	19	0	0	0	25,900
MyBB 1.8.1	947	6,034	532	1,502	27	2	442	9,486
MyBB 1.8.4	891	5,339	725	150	28	2	607	7,742

We first evaluated how well the crawlers cover a web application. A measure for the coverage is the code coverage, i.e., the number of lines of code that have been exercised by the testing tool. While this measure is adequate for code-based testing tools, it may not be for web application crawlers. As web crawlers operate in a black-box setting, it has a limited visibility of the web application. In addition, web crawlers do not fuzz input fields, but they rather use a user-provided list of inputs. As a result, it may not exercise all the branches, thus, leaving unvisited significant portion of the web application. An alternative measure can be the number of URLs a crawler can extract.

A web crawler is a component which provides a web scanner with the URLs to be tested. As the goal of a web scanner is the detection of web vulnerabilities, the second aspect to evaluate is the detection power. The detection power can be measured in terms of the number of reported vulnerabilities. Unfortunately, such a metric may not be fair. Prior research has shown that this type of evaluation is not an easy task. Web scanners do not support the same classes of vulnerabilities

and they may differentiate the target vulnerabilities. In result, the number of discovered vulnerabilities cannot be comparable among the different crawlers. For this reason, in this paper we limited our comparison to a specific class of vulnerabilities, i.e., reflected XSS. Second, the number of reported vulnerabilities may contain false positives. A false positive happens when the scanner reports the existence of a vulnerability but the vulnerability does not acutally exist. The number of false positives also measures the accuracy of the web scanner and indicates whether a scanner adequately verifies if the observed behavior qualifies as a vulnerability.

Case Studies. We performed our assessment using 13 popular web applications. These applications include three content management systems (i.e., Joomla 3.4.1, Modx-CMS 2.02.14, and Nibbleblog 4.0.1), a blogging tool with plugins (i.e., WordPress 3.7 and 4.0.1, and Tidio 1.1), discussion forum software (i.e., MyBB 1.8.01 and 1.8.04, and phpNN 3.0.12), photo gallery applications (i.e., Gallery 2.7.1 and Piwigo 2.7.1), cloud storage applications (i.e., OwnCloud 4.0.1 and 8.0.3), and wiki web application (i.e., MediaWiki 1.24.2). Among these, the following five web application are already known to be vulnerable to reflected XSS: Modx-CMS, MyBB 1.8.01, phpBB, Piwigo, and OwnCloud 4. These web applications vary in size, complexity, and functionality. We set up these web applications on our own servers. Each web application was installed in a virtual machine. We reset the state of the virtual machines upon each test.

Results. We divide the evaluation results into two parts. First, we investigate the diversity of events that jÄk has found and measure the coverage of the crawlers. Second, we assess how well jÄk performs in detecting XSS vulnerabilities as compared to other scanners.

Coverage — Table 2 shows the number of unique event-handler registrations extracted by jÄk. The number of events are shown for each web application, grouped by event category, i.e., device-dependent input events (DDI), device-independent input events (DII), Change events (Chg), API events, Error events, and custom errors. These events are extracted via the dynamic analysis of the client-side JavaScript program of the case studies.

Table 2 shows that web applications can rely on JavaScript events in a moderate way, i.e., Nibbleblog, or more heavily, i.e., Mediawiki. Most of the registered event handlers are of the device input and UI categories. Just these events amount to 68 % of all events, whereas UI events amount to 22 %.

Next we show asses whether jÄk outperforms existing crawlers in terms of coverage. To this end, we measure the number of unique URL structures each crawler found. The URL structure is a URL without the query string values. Table 3 shows the results, excluding all URLs for static and external resources. Numbers in bold mark the tool that extracted the highest number of URL structures. The symbol * indicates that the results of W3af and Skipfish do not take into account invalid URLs that have been found via URL forgery (as explained

Table 3. Coverage of the web applications in terms of unique URL structures, excluding linked and static resources, e.g., CSS documents, external JS files, and images. The symbol * indicates the numbers which do not count URL forgery by W3af and Skipfish.

Web Apps.	jÄk	Crawljax	W3af	Wget	Skipfish
WP	**21**	15	17*	34	17*
Gallery	**180**	7	35	33	24
phpBB	**50**	11	44	27	27
Joomla	4	5	**7***	3	5*
Tidios w/ WP	166	21	**251***	218	35*
NibbleBlog	**7**	6	**7***	5	**7***
OwnCloud 8	**98**	2	54*	44	14*
OwnCloud 4	**80**	–	58	10	61
Piwigo	**277**	15	58	13	24
Mediawiki	**1,258**	24	480	265	776*
ModX	**57**	2	21	41	34*
MyBB 1.8.1	**152**	22	95	131	126
MyBB 1.8.4	**152**	12	92	135	128
Total	**2502**	142	1219	959	1278

later). jÄk extracted the highest number of unique URL structures in 10 applications. In one application, i.e., Nibbleblog, jÄk, W3af, and Skipfish extracted the same number of URL structures. In the remaining two web applications, W3af extracted the highest number of web applications. In the case of Joomla, W3af extracted 3 URL structures more than jÄk, whereas in the case of Tidios, W3af extracted 251 URLs against 166 of jÄk.

To interpret these results qualitatively, we sought to assess to what extent the surfaces explored by each tool relate to the one explored by jÄk. A way to measure this is to analyze the URLs extracted by jÄk and each of the other tools, and to calculate the complement sets. These two sets will contain the following URLs. The first set contains URLs that are extracted by jÄk and missed by each of the other crawlers. The second set contains the URLs that are not discovered by jÄk but are extracted by the other tool. The number of URLs in each of these sets is shown in Table 4. When compared with the other tools, on average, jÄk explored a surface of the web applications which is 86 % larger than the one of the other tools. Then, the amount of surface which jÄk did not explore range from 0.5 % of Crawljax to 22 % of Skipfish.

Table 4. Unique URLs discovered only by jÄk (+) and missed by jÄk (-).

Groups	Crawljax	W3af	Wget	Skipfish
Surf. discovered only by jÄk	+98%	+85%	+70%	+90%
Surf. missed by jÄk	-0.5%	-18%	-20%	-22%

To further understand the potential misses by jÄk, we manually inspected a random sample of 1030 (15 %) of the URLs that are not discovered by jÄk. We were able to identify eight classes of URLs, as shown in Table 5. URL forgery refers to URLs which are not present in the web application but are forged by the crawler. The vast majority of the URLs that jÄk "missed", i.e., 75 % of the URLs, are URLs that were forged by W3af and Skipfish. Forging means that these tools implement a crawling strategy which attempts to visit hidden parts of the web application. Starting from a URL, they systematically submit URLs in which they remove parts of the path. For example, W3af derives from URLs like http://foo.com/p1/p2/p3, other URLs, i.e., http://foo.com/p1/p2, http://foo.com/p1/, and http://foo.com/. It is important to notice that these URLs are not valid URLs of the web application. For this reason, we corrected the results in Table 3 by deducting the percentage of forged URLs. Next, the class static resources include style-sheet documents or external JS files with a different document extension, e.g., `.php`. This is an error introduced by our URL analysis which failed in recognizing these documents as static. The third class of URLs (5.34 %) is the one of unsupported actions such as form submission during crawling. Then, the fourth class contains URLs that were not extracted because they belong to a user session different from the one used by jÄk. This may be solved by using jÄk in parallel with multiple user credentials. The fifth class contains URLs that are due to bugs both in jÄk and in Skipfish. The sixth class contains URLs that are generated while crawling. We have two types of these URLs: URLs with timestamps and URLs generated by, for example, creating new content in the web application. 1,36 % of the URLs, we could not find the origin of the URL nor the root cause. Finally, 1,26 % of the URLs are of W3af that does not implement a depth-bounded crawling and thus might crawl the applications deeper than other crawlers.

Detection — Finally, we measure how the improved crawling convergence translates into the detection of XSS vulnerabilities in the 13 web applications. For these tests, we had to exclude Wget and Crawljax, as they are pure crawlers and as such cannot discover vulnerabilities.

jÄk discovered XSS vulnerabilities in three of the five web applications, i.e., phpBB, Piwigo, and MyBB 1.8.1. However, jÄk could not find known vulnerabilities in OwnCloud 4 and ModX. Manual analysis revealed that the vulnerability as described in the security note of OwnCloud 4 is not exploitable. For ModX, jÄk could not discover the URL. The URL is added in the DOM tree by an event handler. jÄk correctly fires the events, but the code of the handler is not executed because it verifies that the target is an inner tag. This shortcoming is the same that cause to fail the test in the `C5` class of Table 1. In a regular browser, due to the implicit rules for the propagation of events, the user will click on the inner tag and the outer one will be executed. As a future work, we plan to reproduce the event propagation as implement by regular browsers.

The other tools detected only known vulnerabilities in MyBB, and had issues with false positives. Both W3af and Skipfish detected the XSS vulnerability in MyBB 1.8.1. Furthemore, in Mediawiki W3af reported 49 XSS vulnerabilities

and Skipfish one vulnerability, respectively. However, in both cases, these were false positives. Finally, Skipfish reported 13 false positives in Gallery. In our experiments, jÄk did not report any false positive. This is the result of using dynamic analysis for the detection of XSS attacks: if an attack is successful, the test payload is executed and visible in the dynamic trace.

Table 5. Origin of the URLs that were not discovered by jÄk

URL Origin	URLs	Fraction
URL Forgery	774	75.15%
Static resources	57	5.53%
Unsupp. action	55	5.34%
User session mgmt.	53	5.15%
Bugs	47	4.56%
New content	17	1.65%
Unknown	14	1.36%
Beyond max depth (W3af)	13	1.26%
Total	**1030**	**100,00%**

6 Related Work

In this section we review works closely related to our paper. We focus on two areas: analysis of existing web application scanners, and novel ideas to improve the current state of the art of scanners.

Bau et al. [15] and Doupé et al. [16] presented two independent and complementary studies on the detection power of web application scanners. Both works concluded that while web scanners are effective in the detection of reflected XSS and SQLi, they still poorly perform in the detection of other classes of more sophisticated vulnerabilities. According to these works, one of the reason of these results is the lack of support of client-side technology. Furthermore, Doupé et al. explored in a limited way the problem of web application coverage focusing on the capability of scanners to perform multi-step operations. As opposed to these works, in this paper we mainly focused on the problem of the coverage of web applications and detailed the shortcomings of four web application scanners.

Recently, there have been new ideas to improve the state of the art of web application scanner. These works included the support of client-side features and explored the use of reasoning techniques together with black-box testing. The most notable of these works are the *state-aware-crawler* by Doupé et al. [14], *Crawljax* by Mesbah et al. [2], *AUTHSCAN* by Guangdong et al. [17], and *SSOScan* by Zhou et al. [18]. State-aware-crawler proposed a model inference algorithm based on page clustering to improve the detection of higher-order XSS and SQLi. However, this technique focus mainly on the detection of state-changing operations and it does not take into account the dynamic features of

client-side programs. Similarly, Crawljax proposed a model inference technique based on "user-clickable areas" in order to crawl hidden parts of AJAX-based web applications. However, Crawljax uses static heuristics that do not satisfactorily cover the dynamic interaction points between the user and the UI. As opposed to Crawljax, jÄk does not rely on these heuristics and uses a technique which can detect the registration of event handlers via function hooking. Finally, AUTHSCAN and SSOScan are black-box testing tools that focus on the Web-based Single Sign-On functionalities integrated in web applications. AUTHSCAN extends the classical design verification via model checking with the automatic extraction of formal specifications from HTTP conversations. SSOScan is a vulnerability scanner that targets only Facebook SSO integration in third-party web applications. Neither of the two tools is a web application scanner, and they do not support crawling web applications. As opposed to jÄk, the focus of these tools is on improving the detection power of security testing tools. Nevertheless, the proposed testing technique may be integrated into jÄk to detect other classes of vulnerabilities.

A work closely related to our approach is Artemis [19]. Artemis is a JavaScript web application testing framework which supports the generation and execution of test cases to increase the client-side code coverage. Starting from an initial input (e.g., event), Artemis explores the state space of the web application by probing the program with new inputs. Inputs are generated and selected by using different strategies in order to maximize, e.g., code branches or number of read/write access of object properties. At each step, Artemis resets the state of the client and server side to a known state and continues the exploration. From the angle of input generation, Artemis and our approach shares common points. For example, both approaches explore the client side by firing events and observing state changes. However, Artemis and our approach differ on the assumption of the availability of the server side. While Artemis assumes complete control of the state space of the server side, our approach does not make this assumption and targets the exploration of a live instance of the server side.

7 Conclusion

This paper presented a novel technique to crawl web applications based on the dynamic analysis of the client-side JavaScript program. The dynamic analysis hooks functions of JavaScript APIs to detect the registration of events, the use of network communication APIs, and find dynamically-generated URLs and user forms. This is then used by a crawler to perform the next action. The crawler creates and builds a navigation graph which is used to chose the next action. We presented a tool jÄk, which implements the presented approach. We assessed jÄk and four other web-application scanners using 13 web applications. Our experimental results show that jÄk can explore a surface of the web applications which is about 86 % larger than the other tools.

Acknowledgements. This work was supported by the German Ministry for Education and Research (BMBF) through funding for the project 13N13250, EC SPRIDE

and ZertApps, by the Hessian LOEWE excellence initiative within CASED, and by the DFG within the projects RUNSECURE, TESTIFY and INTERFLOW, a project within the DFG Priority Programme 1496 *Reliably Secure Software Systems* $-RS^3$.

References

1. Zhou, J., Ding, Y.: An analysis of URLs generated from javascript code. In: 2012 IEEE/ACIS 11th International Conference on Computer and Information Science (ICIS), vol. 5, pp. 688–693 (2012)
2. Mesbah, A., van Deursen, A., Lenselink, S.: Crawling ajax-based web applications through dynamic analysis of user interface state changes. ACM Trans. Web **6**(1), 3:1–3:30 (2012)
3. Urgun, B.: Web Input Vector Extractor Teaser (2015). https://github.com/bedirhan/wivet
4. Hickson, I.: A vocabulary and associated APIs for HTML and XHTML (2014). http://dev.w3.org/html5/workers/
5. van Kesteren, A., Gregor, A., Ms2ger, Russell, A., Berjon, R.: W3C DOM4 (2015). http://www.w3.org/TR/dom/
6. The Python Software Foundation: Python (2015). https://www.python.org/
7. Apple Inc.: The WebKit Open Source Project (2015). https://www.webkit.org/
8. Riverbank Computing Limited: PyQt - The GPL Licensed Python Bindings for the Qt Application Framework (2015). http://pyqt.sourceforge.net/
9. Google Inc.: V8 JavaScript Engine (2015). https://code.google.com/p/v8/
10. Mozilla Foundation: SpiderMonkey (2015). https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey
11. Zalewski, M.: Skipfish (2015). https://code.google.com/p/skipfish/
12. Riancho, A.: w3af: Web Application Attack and Audit Framework (2015). http://w3af.org/
13. Nikšić, H., Scrivano, G.: GNU Wget (2015). http://www.gnu.org/software/wget/
14. Doupé, A., Cavedon, L., Kruegel, C., Vigna, G.: Enemy of the state: a state-aware black-box vulnerability scanner. In: Proceedings of the 2012 USENIX Security Symposium (USENIX 2012), Bellevue, WA (2012)
15. Bau, J., Bursztein, E., Gupta, D., Mitchell, J.: State of the art: automated black-box web application vulnerability testing. In: 2010 IEEE Symposium on Security and Privacy (SP) (2010)
16. Doupé, A., Cova, M., Vigna, G.: Why Johnny can't pentest: an analysis of black-box web vulnerability scanners. In: Kreibich, C., Jahnke, M. (eds.) DIMVA 2010. LNCS, vol. 6201, pp. 111–131. Springer, Heidelberg (2010)
17. Guangdong, B., Guozhu, M., Jike, L., Sai, S.V., Prateek, S., Jun, S., Yang, L., Jinsong, D.: Authscan: Automatic extraction of web authentication protocols from implementations. In: 2013 Annual Network and Distributed System Security Symposium (NDSS). The Internet Society (2013)
18. Zhou, Y., Evans, D.: Ssoscan: automated testing of web applications for single sign-on vulnerabilities. In: 23rd USENIX Security Symposium (USENIX Security 2014), pp. 495–510. USENIX Association, San Diego, CA (2014)
19. Artzi, S., Dolby, J., Jensen, S.H., Møller, A., Tip, F.: A framework for automated testing of javascript web applications. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, pp. 571–580. ACM, New York, NY, USA (2011). http://doi.acm.org/10.1145/1985793.1985871