Race-Driven UI-Level Test Generation for JavaScript-Based Web Applications

Martin Billes

Department of Computer Science, TU Darmstadt, Germany

martin.billes@cased.de

Abstract

Due to its asynchronous, event-driven nature, JavaScript, similar to concurrent programs, suffers from the problem of data races. Past research provides methods for automatically exploring a web application to generate a trace and uses an offline dynamic race detector to find data races in the application. However, the existing random exploration techniques fail to identify races that require complex interactions with the application. While more sophisticated approaches to explore websites exist, these are not targeted towards finding data races. We conduct a study of data race bugs in open source software that shows most data race bugs are related to AJAX requests. Motivated by these findings, we present an approach for UI-level test generation which explores a website with the goal to find additional data races.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Testing tools

Keywords AJAX, Automated testing, Data race, Event-driven, JavaScript, Web applications

1. Introduction

Within the context of parallel software, data races are two accesses to the same memory location, with at least one of them being a write access, that do not have an enforced order. JavaScript has a single-threaded execution model, and the web platform extends it with an event-driven model. Events are triggered by the system or the user and execute from the event queue. The order in which these events execute can be non-deterministic. The code in different event handlers thus can be considered unordered.

SPLASH Companion'15, October 25–30, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3722-9/15/10
http://dx.doi.org/10.1145/2814189.2815364

Several automatic ways to detect data races in web applications have been proposed. They expand on existing work to detect data races in concurrent software [2, 6]. Raychev et al. [5] build a race detector named EVENTRACER (based on earlier work [4]), which works on a single execution trace of the web application. EVENTRACER is limited by the automated UI-level web application exploration technique used with the tool, which prevents it from reaching races that require a complex set of user interactions. The large number of reported races for complex websites also makes it hard to tell which races are critical. Mutlu et al. [3] proposed a race detector that focuses on races on persistent browser state, such as cookies and `localStorage`, limiting the number of reported races, but it is affected by similar automatic exploration limitations.

The existing data race detection techniques depend on an exhaustive initial execution trace. Other than manual exploration, the most common approach used to generate the execution trace is random exploration of the website [3–5], which generally involves loading the page, entering values into input fields and clicking on links and buttons.

2. Bug Study

To better understand which races matter in practice, we manually inspect bugs in the bug trackers of several open source projects. We search the bug trackers for JavaScript bugs, without looking for particular keywords. We are able to identify 11 data race bugs across the projects WordPress, owncloud and jquery-pjax, of which we are able to locally reproduce 6. Out of the 11 bugs, all but two have at least one `XMLHttpRequest` (XHR) callback participating in the race. In seven cases, two competing XHR callbacks are responsible for the race.

Only one bug can be triggered by simply loading the page. The other bugs require complicated user input scenarios such as triggering a sequence of requests while having a slow connection to the server, or typing into an input field and then clicking a specific button without focusing any other DOM element in between.

Consider the example in Fig. 1 adapted and stripped down from WordPress bug 17936[1]. The application maintains a

```
1  document.getElementById("category-add").
       addEventListener("click",function(event) {
2    var new_category = document.getElementById("
         new-category").value;
3    if (mirroredArray.indexOf(new_category) !==
         -1)
4      return;
5    var xhr = new XMLHttpRequest();
6    xhr.open("POST", "/add");
7    xhr.onreadystatechange = function() {
8      if (xhr.readyState === 4 && xhr.status ===
           200)
9        mirroredArray.push(xhr.responseText);
10   };
11   xhr.setRequestHeader("Content-Type", "text/
         plain");
12   xhr.send(new_category);
13 }, false);
```

Figure 1. JavaScript code for synchronizing an array of categories with a server via AJAX.

`mirroredArray` of categories of items and the UI contains a text input and a button for adding a new category. Clicking the button checks whether the new category name already exists in the `mirroredArray`, and if not, submits the text input's value to the server via an XHR. The server saves the category and sends its name back to the client, where it is added to the `mirroredArray`.

Between sending the request and receiving the response, the user may click the button again, which will result in another request for the same category name being sent to the server, violating the application's intended consistency. There is a data race on the memory location of `mirroredArray` between the two events `click` (line 3) and `readystatechange` (line 9).

The data race cannot be easily detected by random exploration, especially of the kind which tries to trigger every enabled event only once (as in [5]), since it requires clicking on the relevant button twice. Automatic exploration tools such as ARTEMIS [1] which try to maximize code coverage will also not try to trigger an event handler a second time after its code has already been covered.

Overall, the bug study results show that there is a need for automated techniques that require a complex set of user interactions to trigger data races.

3. Advancing Automatic Exploration

We are designing a systematic automatic exploration technique for the domain of finding data races in web applications. The test generator queries the list of available events to trigger in a web application and then chooses to trigger one of those events based on a priority function. This creates an execution trace which we use in the data race detector to confirm races. Using a test-generation approach that is specifically targeted towards data races promises to produce more valuable initial traces for data race detection tools such as EVENTRACER to analyze.

Our approach keeps track of the set of memory locations that every event has read or written to. This information can be used to detect that a top-level UI event such as a button click and its dependent XHR callback both access the memory location. If there are no protective measures in place that prevent a second triggering of the `click` event handler, this overlap of read/write sets can lead to a data race. To test for these, the test generator explores a click on the button two more times, while delaying the response of the XHR callback using network traffic shaping.

A data race of this kind can also happen if the two `click` (or similar) events which the user can trigger are not attached to the same, but to different DOM elements. For this case, identifying the elements which have the same event handler registered is necessary.

In the example in Fig. 1, the algorithm first explores a click on the button element. It detects that both the `click` handler and the `readystatechange` handler access `mirroredArray`. To check for a data race, it clicks on the button two more times, while preventing the XHR response to arrive immediately after the first of those clicks. This explores the data race properly.

We are currently implementing the automatic exploration tool using a Firefox plugin. The generated sequence of UI events can then be fed into a state-of-the-art offline race detector such as EVENTRACER.

[1] See `https://core.trac.wordpress.org/ticket/17936`.

Acknowledgments

This research is supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, and by the German Research Foundation (DFG) within the Emmy Noether Project 'ConcSys'.

References

[1] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *ICSE*, pages 571–580, 2011.

[2] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *ACM Sigplan Notices*, volume 44, pages 121–133, 2009.

[3] E. Mutlu, S. Tasiran, and B. Livshits. Detecting JavaScript races that matter. 2015.

[4] B. Petrov, M. T. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *PLDI*, pages 251–262, 2012.

[5] V. Raychev, M. T. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *OOPSLA*, pages 151–166, 2013.

[6] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.