Practical AJAX Race Detection for JavaScript Web Applications

Christoffer Quist Adamsen
Aarhus University
Denmark
quist@cs.au.dk

Anders Møller
Aarhus University
Denmark
amoeller@cs.au.dk

Saba Alimadadi
Northeastern University
MA, USA
saba@northeastern.edu

Frank Tip
Northeastern University
MA, USA
f.tip@northeastern.edu

ABSTRACT

Asynchronous client-server communication is a common source of errors in JavaScript web applications. Such errors are difficult to detect using ordinary testing because of the nondeterministic scheduling of AJAX events. Existing automated event race detectors are generally too imprecise or too inefficient to be practically useful. To address this problem, we present a new approach based on a lightweight combination of dynamic analysis and controlled execution that directly targets identification of harmful AJAX event races.

We experimentally demonstrate using our implementation, AjaxRacer, that this approach is capable of automatically detecting harmful AJAX event races in many websites, and producing informative error messages that support diagnosis and debugging. Among 20 widely used web pages that use AJAX, AjaxRacer discovers harmful AJAX races in 12 of them, with a total of 72 error reports, and with very few false positives.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

event race detection, JavaScript, dynamic analysis

ACM Reference Format:
Christoffer Quist Adamsen, Anders Møller, Saba Alimadadi, and Frank Tip. 2018. Practical AJAX Race Detection for JavaScript Web Applications. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3236024.3236038

1 INTRODUCTION

Millions of JavaScript web applications use AJAX for client-server communication. AJAX is today a commonly used term that describes uses of the `XMLHttpRequest` (abbreviated XHR) API that all modern browsers support. This API enables JavaScript programs running in browsers to send HTTP requests to a server based on a user's input, and receive responses that are then used to update the UI. A prominent example is the autocomplete feature at `google.com` where suggestions are provided as soon as the user starts entering a search term.

To ensure a smooth user experience, AJAX communication is usually asynchronous, meaning that a user can continue interacting with the page and that more JavaScript code can be executed between the time that the HTTP request is sent and when the response is received. Additionally, it is possible to perform multiple AJAX interactions simultaneously. This may cause nondeterminism when the same sequence of user events leads to different behaviors depending on the order in which the AJAX responses and other events are processed. Often, this leads to errors that are missed by ordinary testing when programmers are insufficiently aware of the many possible interleavings of events. For the end user, the consequences of such errors typically range from minor functionality glitches to misleading inconsistencies in the UI.

The fact that the JavaScript execution model is susceptible to so-called event races is well known [13, 23]. Many techniques have been developed to detect and prevent event race errors and their harmful consequences [1, 2, 10, 14, 19–21, 24–26]. However, none of those existing techniques are capable of detecting harmful event race errors that involve AJAX with sufficient precision and performance to be practically useful. In particular, EventRacer [21] reports too many benign races and may miss harmful ones [2], R^4 [14] relies on stateless model checking, which does not scale well and requires complex browser instrumentation, and InitRacer [2] only detects event races that appear during the initialization of a web application, not those that involve AJAX later in the execution.

The goal of our work is to provide an automated event race detector that is practical for event races that involve AJAX. Specifically, such a tool should be able to detect AJAX event races that have observable effects, without reporting a large number of spurious or harmless races. Also, it should not require browser instrumentation, which is platform-specific and difficult to maintain as browsers evolve, and it should be fast and produce informative error messages that facilitate debugging.

We present an approach that meets these requirements, inspired by the ideas of *adverse execution* used by InitRacer [2] and *controlled execution* used by EventRaceCommander [1]. Our approach is based on the key observation that JavaScript developers typically test their code using networks and servers that are fast and reliable,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236038

so in their tests AJAX is effectively synchronous, meaning that the AJAX request-response pairs are essentially atomic, without other events occurring in between. This observation allows us to establish a notion of "expected" event schedules as those where an AJAX response event handler e_{resp} executes immediately after the event handler e_{req} that sent the request. In contrast, any schedule where another event handler e is scheduled between e_{req} and e_{resp} can be regarded as less likely to be exercised during ordinary testing. An *AJAX event race* occurs if the effects of e conflict with the effects of e_{resp}. The idea of adverse execution is to systematically expose a program to adverse conditions and compare the result with the normal behavior. In our case, schedules where AJAX is processed synchronously define the normal expected behavior, and adverse conditions are situations where the network or server is slow or unreliable allowing other events to interfere.

Our approach consists of two phases. The first phase dynamically monitors an execution of a web application, with the purpose of identifying (1) user event handlers that have conflicting AJAX response event handlers, and (2) information about which event handlers may be reordered. This initial execution may be driven by a human user, an automated testing tool, or a pre-existing test script, similarly to other dynamic race detectors. For each user event handler u that has been observed, an *event graph* $\mathcal{G}_u$ is generated that captures relevant information about the events that have been triggered either directly or indirectly by u. For example, clicking on a button may create a timer event that leads to an AJAX request that, in turn, triggers an AJAX response event, which finally updates the UI. The second phase uses these event graphs to plan a series of tests. Each test simulates two event schedules, one where AJAX is synchronous and one that simulates adverse conditions as discussed above, and automatically compares screenshots of the resulting web pages. Observable differences are reported along with detailed information about the event schedules that gave rise to them. To control the scheduling of event handlers when executing the tests, we use an event controller mechanism inspired by EventRaceCommander in which nondeterminism is restricted by selectively postponing the execution of event handlers.

We evaluate AjaxRacer using 20 web pages from 12 large and widely used web applications. The results show that the approach is effective in detecting AJAX races in real settings. AjaxRacer generates 152 tests, of which 65 reveal harmful races among 12 of the web pages, and only seven reports are false positives. We additionally demonstrate the usefulness of AjaxRacer's comprehensive web-based reports for understanding the detected AJAX races and diagnosing their root causes.

In summary, this paper makes the following contributions:

- We define a notion of *event graphs* that captures relevant information about effects and orderings of event handlers, relative to a given initial execution.
- We present a two-phased approach for automatically detecting harmful AJAX event races in JavaScript web applications. The first phase performs a dynamic analysis for computing event graphs; the second phase executes the generated tests under different event schedules and determines if observably different results appear.
- We describe the open-source tool AjaxRacer, which implements the approach.

```
1  function fetchJSONFromURL(url, callback) {
2    var xhr = new XMLHttpRequest();
3    xhr.open('GET', url, true);
4    xhr.onreadystatechange = function () {
5      if (xhr.readyState == XMLHttpRequest.DONE && xhr.status == 200) {
6        callback(JSON.parse(xhr.responseText));
7      }
8    };
9    xhr.send(null);
10 }
```

Figure 1: AJAX example that demonstrates how a web application can fetch a JSON object from a server.

- We present experimental results showing AjaxRacer to be effective at detecting AJAX races in real-world web applications, that it reports few false positives, and that it provides insightful explanations that are helpful to developers.

2 BACKGROUND ON AJAX

AJAX (Asynchronous JavaScript and XML) is a technology that enables web applications to exchange data asynchronously with a server without imposing page reloads, which enables rich and responsive client-side web applications.

Figure 1 illustrates how a web application can retrieve a JSON object asynchronously from a server using the `XMLHttpRequest` (XHR) API.[1] To send an XHR request, a web application first needs to construct an XHR object (line 2) and initialize the object by calling the `open` method with the relevant HTTP method and URL (line 3). The `open` method takes as optional arguments a boolean that specifies if the request should be asynchronous (defaults to `true`) and credentials for authentication purposes. When the XHR object has been initialized, the AJAX request can be sent by calling the `send` method, optionally with data for the body of the request (line 9).

Each XHR object goes through several phases during the lifecycle of the corresponding request. The current state of an XHR object can be accessed at any time by reading its `readyState` property. This state indicates (among others) if the request has been sent, if the headers and status code have been received from the server, or if the entire response has been received. Each time the state of an XHR object changes, a so-called `readystatechange` event is triggered. Web applications can react to these events by registering an event handler for this event type, as in line 4. The event handler in lines 4–8 explicitly checks that the response has been fully received before it accesses the body of the AJAX response in line 6. XHR involves several other kinds of events, in addition to `readystatechange` events. These include a `load` event when the resource has been loaded, a `timeout` event if the response takes too long, and an `error` event if, for example, the request is blocked by the browser's same-origin policy.

To circumvent the same-origin policy of XHR, many websites instead implement AJAX using JSONP. To get data from a server with that approach, the client code dynamically creates a `script` element with the URL of a script, which is executed when it has

[1] See https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest. The new Fetch API (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) and WebSockets (https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API) provide related functionality. In this paper, we focus on XHR, which is currently the most widely used AJAX API, but the alternatives may be interesting for future work.

www.chevronwithtechron.com/findastation.aspx

```
11 <script src="js/gmap-helper-main-compiled.js"></script>
12 <input id="search" value="Enter Location/ZIP code">
13 <div id="searchBar">
14   <p>FILTER YOUR SEARCH</p>
15   <a onclick="addRemoveFilter('search7');" ...>Car Wash Locations</a>
16   <a onclick="addRemoveFilter('search11');" ...>Diesel Locations</a>
17 </div>
18 <div id="stationResult"></div>
```

www.chevronwithtechron.com/js/gmap-helper-main-compiled.js

```
19 var curGeoObj, filters = [];
20 function addRemoveFilter(filterId) {
21   toggleFilter(filterId);
22   $('#stationResult').html('');
23   searchLocationsNearByJSON();
24 }
25 function searchLocationsNearByJSON() {
26   var url = createURL("webservices/GetStationsNearMe.aspx",
27                       curGeoObj, filters);
28   $.ajax({ url: url, type: "GET", success: parseStationData });
29 }
30 function parseStationData(data) {
31   if (data.status == "ok") {
32     var htmls = [];
33     for (var i = 0; i < data.stations.length; i++) {
34       var html = ...;
35       htmls.push(html);
36     }
37     $("#stationResult").html(htmls.join("<hr />"));
38   } else ...
39 }
```

Figure 2: Motivating example.

been retrieved from the server. For this reason we also need to take dynamically loaded scripts into account.

We distinguish between *user* events (mouse click events, keyboard events, etc.) and *system* events (most importantly, AJAX response events and timer events). After the web page has been loaded and initialized, every system event is triggered either directly or indirectly by a user event. Each such system event can thus be associated uniquely with a user event; we say that the system event is *derived* from that user event.

AJAX is one of the key ingredients of modern web applications. However, it also introduces complexities in the execution of web applications. In particular, there are no guarantees regarding the exact timing and order of arrival of AJAX requests at the server, nor of the corresponding AJAX response events at the client. The user controls the ordering of user events, but the execution of system events is to some extent nondeterministic. Borrowing terminology from concurrency in multi-threaded settings, a *schedule* fixes the nondeterministic choices relative to a given sequence of user events. As a consequence of this nondeterminism, event race errors may occur in production web applications when the order of events in the execution differs from the ones observed during testing.

3 MOTIVATING EXAMPLE

Figure 2 shows a snippet of HTML and JavaScript code from www.chevronwithtechron.com/findastation.aspx. This web page allows the user to search for gas stations in a given area, and to filter these gas stations based on various criteria. For example, the user can search for gas stations that have a car wash by clicking on the "Car Wash Locations" button defined in line 15, which causes

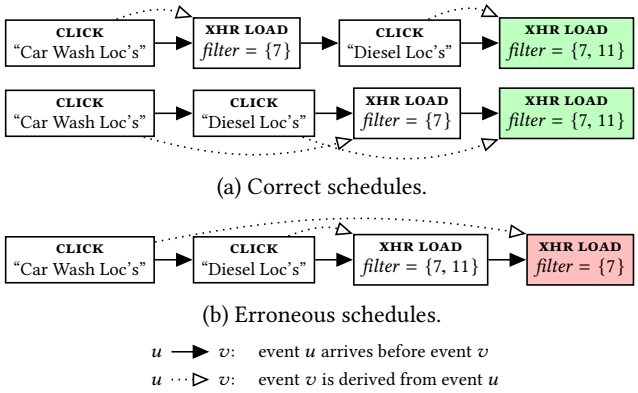


Figure 3: Possible interleavings in the motivating example.

the JavaScript function addRemoveFilter in line 20 to execute. This function updates the set of filters that have been selected by the user (line 21), clears the contents of the HTML element that presents the list of gas stations to the user (line 22), and finally invokes the function searchLocationsNearByJSON (line 23) to retrieve the list of gas stations from the server according to the search query provided by the user (lines 26–28). By the time the server response arrives, an AJAX response event fires, causing the event handler parseStationData (line 30) to execute. This function constructs a snippet of HTML for each gas station in the server response (lines 33–36), and then updates the UI using these snippets (line 37).

The example web page exhibits an AJAX event race when the user selects more than one criterion. Consider what happens when the user clicks on the "Car Wash Locations" button and subsequently on the "Diesel Locations" button. Each of these click events causes an AJAX request to be sent in line 28. The corresponding AJAX response events arrive asynchronously and without a predetermined order, so either may be processed first. If the AJAX response corresponding to the click on the "Car Wash Locations" button is processed first, then the web page works correctly, since the subsequent AJAX response event (corresponding to the click on the "Diesel Locations" button) simply updates the UI with the gas stations that have a car wash *and* diesel. However, if the AJAX responses arrive in the opposite order, then the AJAX response event corresponding to the click on "Car Wash Locations" results in an inconsistent state: the filters "Car Wash Locations" and "Diesel Locations" are both selected (and highlighted in the UI), but the list of gas stations in the UI only shows those stations that have a car wash, but not necessarily diesel.

Figure 3a illustrates two schedules that lead to correct behavior for the example user event sequence with the two button clicks. (For simplicity, it only shows a subset of the actual events that occur.) In one schedule, the AJAX response event derived from the first button click occurs before the second button click, and vice versa in the other schedule. In both cases, the AJAX response event derived from the second button click comes last. Figure 3b shows a third schedule for the same user event sequence. In this case, the AJAX response events arrive out of order, which results in the error.

The event handler for the "Car Wash Locations" button not only conflicts with the event handler for the "Diesel Locations" button, but also conflicts with itself. In particular, it is possible to expose an error that is similar to the one described above, by triggering two

simultaneous click events on the "Car Wash Locations" button. If the AJAX responses arrive out of order, the markers on the map are inconsistent with the selected filters. The technique we describe in the following sections finds both these errors.

For an event race error detection technique to be practical, it is not sufficient for it to detect errors and produce useful error messages; it is also important that it does not report too many false positives. Predictive event race error detectors like EventRacer generally report many races that are infeasible or harmless [2, 19, 25]. This is particularly problematic when web application programmers carefully use ad-hoc synchronization to avoid race errors. For this reason, our technique is designed so that it only reports event race errors that can be witnessed by concrete schedules that exhibit visible differences in the browser.

4 THE AJAXRACER TECHNIQUE

Our technique comprises two phases. Phase 1 generates event graphs that can be used to identify pairs of user events that are likely to be involved in an observable AJAX event race. Phase 2 examines, for each such pair of events, whether or not an observable AJAX event race actually exists.

4.1 Phase 1: Generating Event Graphs

Phase 1 is seeded by a sequence of user events, similar to other dynamic race detectors [14, 19, 21]. This sequence can be obtained by a single manual execution of the web application, or using an automated crawler [3, 17]. AjaxRacer loads the (instrumented) web page in the browser and waits until it has been fully initialized (meaning that the HTML has been parsed, its scripts have been executed, and there are no pending system events; see Section 5 for details). It then triggers the user events in the sequence one by one, in each step awaiting a quiescent state where no system events are pending, until the next user event is triggered. With such a controlled execution, it is easy to determine from which user event each system event is derived, and we reduce the risk of interference.[2]

For each user event u, AjaxRacer generates a *trace* τ_u by monitoring the execution of u and its derived system events. A trace is a sequence of operations of the following kinds:

- **fork**$[v, w, k]$ models the fact that an event v creates a new system event w of kind k to be dispatched later. For example, **fork**$[v, w,$ *XHR load*$]$ means that v performs an XHR request and w is the associated *XHR load* event, and **fork**$[v, w,$ *timeout*$]$ means that v sets a timer using `setTimeout` and w is the associated timeout event.
- **join**$[v, w]$ specifies that event w cannot occur before event v. Every XHR request creates several *XHR readystatechange* events and an *XHR load* event, and we use **join** to model the ordering constraints on those events.
- **mutate-dom**$[v, x, y, w, h]$ models that event v has modified the HTML DOM, where the parameters x, y, w, h specify the position and size of the affected bounding box on the screen.[3]

[2] As an example, if we did not wait between the user events but triggered them without any delay, an unfinished XHR interaction initiated by one user event might be aborted by an XHR interaction initiated by another user event.

[3] Other effects, for example involving web storage or cookies [19], can be modeled as variants of this operation.

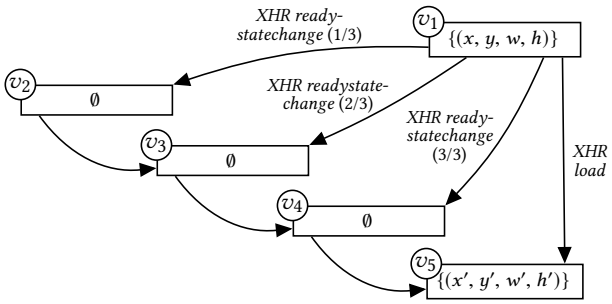


Figure 4: The event graph for a click event on the "Car Wash Locations" button from Section 3. The click event leads to three *XHR readystatechange* events and an *XHR load* event.

Compared to the notion of event actions in EventRacer [21], the key differences are that (1) we generate one trace per user event rather than one global trace, and (2) we use a different model of memory accesses where we consider the effects of HTML DOM write operations on the pixels on the screen instead of low-level read/write operations.

From each trace τ_u, AjaxRacer now generates an *event graph* $\mathcal{G}_u$. An event graph is a directed graph $\mathcal{G}_u = (N, E, \ell)$ where each node $v \in N$ is an event, which is either u itself or an event derived from u, and where the edges E represent constraints on the event order:[4] Each operation **fork**$[v, w, k]$ in τ_u gives rise to a labeled edge $v \xrightarrow{k} w \in E$, and each operation **join**$[v, w]$ in τ_u gives rise to an unlabeled edge $v \rightarrow w \in E$. The component ℓ annotates each node with a set of bounding boxes according to the HTML DOM modifications: for each operation **mutate-dom**$[v, x, y, w, h]$, the bounding box (x, y, w, h) is included in $\ell(v)$. The event graph thus describes the HTML DOM modifications made by the user event u and all its derived system events. We will refer to the user event u as the (unique) *root* of $\mathcal{G}_u$.

Example. Figure 4 shows a simplified version of the event graph for a click event on the "Car Wash Locations" button from Section 3. The root is the click event itself. Since the `addRemoveFilter` function clears the contents of the HTML element with ID `#stationResult` (line 22), the node annotation contains its bounding box ($x = 280$, $y = 1132$, $w = 1024$, $h = 334$). The bottom-most node represents the *XHR load* event, whose event handler updates the same HTML element, as indicated by the node annotation. □

Our approach targets a scenario in which web application programmers have tested their code using fast servers and networks, and with plenty of time between each user event. In such situations, if a user event u_1 is followed by a user event u_2, it is to be expected that all events derived from u_1 appear before u_2 and all of its derived events. It is less likely that the programmers have encountered executions in which some of the events derived from u_2 appear before some of the events derived from u_1. Such executions are exactly what AjaxRacer aims to explore.

For that purpose, we now define a suitable notion of event conflicts. Let u_1 and u_2 be user events with event graphs $\mathcal{G}_{u_1} = (N_1, E_1, \ell_1)$ and $\mathcal{G}_{u_2} = (N_2, E_2, \ell_2)$, respectively. The two user events

[4] Notice that the event graph captures a happens-before relation in the style of Petrov et al. [20]: $v \preceq w$ if there is a path from v to w.

Algorithm 1: Planning AJAX race tests.

foreach (u_i, u_j) where $i, j \in 1, \ldots, n$ **do**
  if u_i *and* u_j *are potentially AJAX conflicting* **then**
    test (u_i, u_j)
  end
end

u_1 and u_2 are *potentially AJAX conflicting* if there exists an event $v_1 \in N_1$ and an event $v_2 \in N_2$ such that

(1) u_1 and v_1 are separated by an AJAX event, meaning that $\mathcal{G}_{u_1}$ has a path from u_1 to v_1 containing an edge $\xrightarrow{k}$ where the label k is *XHR load* or *script load*, and
(2) a bounding box in $\ell(v_1)$ overlaps with one in $\ell(v_2)$.

The intuition of the first condition is that u_1 triggers an XHR request or loads an external script, which subsequently leads to an event v_1, and the second condition checks whether v_1 may interfere with events derived from u_2.

We say *potentially* conflicting, because the criterion does not guarantee that u_1 and u_2 are simultaneously enabled. For example, u_1 and u_2 may be click events on two different buttons, where the button for u_2 is created by u_1 or one of its derived events. Also, the event handlers may behave differently depending on the schedule, due to, e.g., ad-hoc synchronization. Phase 2, described in Section 4.2, examines whether potential conflicts are realizable.

In principle, some AJAX race errors require more than two user events to manifest. However, in all real-world cases we are aware of, two user events suffice, so we focus on this more common situation.

Example. As mentioned in Section 3, a "Car Wash Locations" button click event not only conflicts with a "Diesel Locations" button click event, but also with itself. The event graph for a click on "Car Wash Locations", as shown in Figure 4, indeed satisfies the conditions for this event to potentially AJAX conflict with itself: there is a path from v_1 to v_5 containing an *XHR load* event, and the bounding box of v_1 overlaps with that of v_5 (in fact, they are identical in this case). This tells us that it may be worthwhile in Phase 2 to test a user event sequence containing two clicks on "Car Wash Locations", with a schedule where the events derived from the second click appear before those derived from the first click. □

4.2 Phase 2: Testing Potential Conflicts

From Phase 1, we have a sequence of user events $u_1, \ldots, u_n$, each described by an event graph, and we know for each pair of user events whether or not they are potentially AJAX conflicting. In principle, AjaxRacer could simply output the resulting pairs of events as warnings to the user, which would be reminiscent of how predictive race detectors work [19, 21]. However, to avoid many false positives and produce more informative error messages, Phase 2 attempts to provoke actual observable race errors, similar to other techniques [2, 10, 14, 25], but using a mechanism specifically designed for AJAX event races.

We perform a set of tests according to Algorithm 1. For each pair of user events (u_i, u_j), one test is created if the two events are potentially AJAX conflicting. Note that we consider all ordered pairs of user events from $u_1, \ldots, u_n$, including those where $i = j$, which is relevant for the previously mentioned example involving multiple clicks on "Car Wash Locations". In practice, relatively few of the event pairs are potentially AJAX conflicting, so the total number of tests performed is usually low (see Section 6).

Algorithm 2: Executing an AJAX race test.

```
   // execute u_i and u_j in 'synchronous' mode
1  reload the web page
2  trigger u_i
3  wait until the events in G_{u_i} have been executed
4  trigger u_j
5  wait until the events in G_{u_j} have been executed
6  s_1 = screenshot
   // execute u_i and u_j in 'adverse' mode
7  reload the web page
8  trigger u_i, and postpone all its derived AJAX events
9  trigger u_j
10 wait until the events in G_{u_j} have been executed
11 allow the events derived from u_i to execute
12 wait until the events in G_{u_i} have been executed
13 s_2 = screenshot
   // decide outcome
14 if s_1 ≠ s_2 then emit error message
```

Algorithm 2 shows how each test is performed. Lines 1–6 simulate a user event sequence where u_i and u_j are performed after the web page has been loaded, using a schedule where all system events derived from u_i appear before those derived from u_j, as if AJAX communication were synchronous. Next, lines 7–13 simulate the same two user events, but this time using an "adverse" schedule where the AJAX events derived from u_i are postponed until after all the events derived from u_j have appeared. After each run, we take a screenshot of the browser contents, and an error is reported if the two screenshots are not identical (line 14).

When attempting to trigger an event (lines 2, 4, 8, and 9), the test aborts without emitting any error message if the event is not enabled because the associated DOM element does not exist or is not visible. This can happen because other events that appear in the Phase 1 execution but not in the Phase 2 executions may have changed the system state, however this is rarely a problem in practice (see Section 6). One pattern is quite common, though: In many web pages, an HTML element (e.g., a menu item) only becomes visible after clicking or hovering over another HTML element. For this reason, we allow the user of AjaxRacer to group such low-level events in the initial event sequence into "macro events" [7], so that AjaxRacer can trigger them together, which increases the chance of the events being enabled.

Each time the web page is reloaded (lines 1 and 7), we wait until it is fully initialized, as in Phase 1. Waiting for derived events to be executed (lines 3, 5, 10, and 12) is also implemented by waiting until the web page becomes idle. In this way, we do not risk waiting for derived events that were observed in Phase 1 but do not occur in this execution, which is reminiscent of the concept of approximate replay in R^4 [14]. Postponing events (line 8) and allowing them to execute (line 11) is implemented using an approach inspired by EventRaceCommander [1].

Example. Continuing the example, an initial user event sequence that contains a single click event u on "Car Wash Locations" and a single click event v on "Diesel Locations" suffices to find both errors described earlier. One test being performed is for the event pair (u, v). This test first executes u followed by v in "synchronous" mode, and then in "adverse" mode. The resulting screenshots are different, so an error is reported. Another test is performed for the event pair (u, u), and again an error is reported because the screenshots differ between the synchronous and the adverse schedules. □

An important difference between AjaxRacer and other event race detectors like EventRacer [21], R^4 [14], and WAVE [10] is that AjaxRacer not only explores different schedules for the system events but also user event sequences that are different from the seed execution. This allows AjaxRacer to detect errors that are missed by the other techniques.

Consider for example a web page with two buttons, A and B. Clicking the A button triggers an XHR request where the XHR load event adds contents to an HTML element, and clicking the B button clears the contents of the HTML element. In this case, a race error appears if the B button is clicked after the A button is clicked but before the XHR load event occurs. If the initial sequence of user events consists of a click on A followed by a click on B, then EventRacer, R^4, WAVE, and AjaxRacer will all find the error. However, if the initial event sequence consists of a click on B followed by a click on A, then EventRacer, R^4, and WAVE do *not* find the error (because they treat user events as being happens-before ordered), but AjaxRacer does find it.

As another example, consider a web page with a single button C where clicking on C triggers an XHR request, and the XHR load event handler writes the server response data into the HTML DOM. A user event sequence that contains a single C click event may cover all the JavaScript code, but it is not enough for EventRacer, R^4, or WAVE to expose the race error that occurs if C is clicked twice and the responses arrive out of order. In contrast, AjaxRacer can find the error, even with a single occurrence of the C click event in the initial event sequence.

5 IMPLEMENTATION

AjaxRacer is implemented as a command-line JavaScript application that takes as input a URL and a user event sequence to analyze, and is available at `http://www.brics.dk/ajaxracer/`.

The implementation uses a proxy server, mitmproxy,[5] to dynamically instrument HTML and JavaScript source files as they are fetched by the browser. The instrumentation wraps all property assignments and DOM API functions that involve event handlers and modifications of the HTML DOM, so that we can intercept the relevant operations at runtime. Dynamically generated code is instrumented by wrapping the built-in functions `eval` and `Function`.

When the proxy is running, AjaxRacer uses the end-to-end testing framework Protractor[6] to load the given URL in Google Chrome via the proxy server, trigger a given sequence of user events (or macro events, as discussed in Section 4.2), store results from the execution, and optionally take a screenshot of the resulting state. These steps are carried out once for Phase 1 and twice for each test that has been planned in Phase 2 (recall Algorithm 2). The screenshots that are captured for each test are compared using the LooksSame library.[7] AjaxRacer ignores a difference at a pixel (x, y), if the adverse mode and synchronous mode executions already differed at (x, y) when the web page finished loading. This mechanism helps to prevent false positives in situations where a server returns slightly different HTML each time. In addition to classifying the two screenshots as identical or not, AjaxRacer also uses the LooksSame library to generate an image where the differences (if any) are highlighted, which is useful for further debugging.

The instrumentation of the web application code allows AjaxRacer to generate a trace for each user event. It also makes it possible to determine when the web application has finished loading (by waiting for the set of pending events to become empty, as explained in Section 4.1), and when the web application becomes idle after a user event has been triggered and processed.

Some web applications never finish loading, in the sense that they continuously react to timer events (e.g., to implement a slideshow that automatically changes every few seconds). AjaxRacer deals with such situations during page loading by deleting timer events with a delay above a given threshold, and by stopping a chain of timer events if the length of the chain reaches some threshold. We have not found cases where this breaks the main functionality of the web application. Because we wait until the web application is entirely idle, the user event handlers triggered by AjaxRacer cannot interleave with code that has been spawned during the loading of the web application. This helps prevent false positives from the screenshot comparison. For example, in the presence of a slideshow, the screenshots taken by AjaxRacer would otherwise depend on the exact timing, and be unsuitable for use as an oracle. GIF animations are another source of nondeterministic results. To combat this issue, AjaxRacer uses its proxy to intercept the loading of GIF images and remove animations.

6 EVALUATION

To assess the effectiveness of our approach, we conducted three experiments to answer the following research questions:

RQ1 (Effectiveness) Does AjaxRacer report AJAX event race errors in real-world web applications? How often do AjaxRacer's warnings identify real errors?

RQ2 (Race characteristics) Do the detected AJAX races exhibit interesting patterns?

RQ3 (Usefulness) Do the generated reports provide informative explanations of the causes and effects of each AJAX event race?

RQ4 (Performance) Is AjaxRacer's performance acceptable?

RQ5 (Comparison with state-of-the-art) How effective is AjaxRacer compared to other tools, most importantly EventRacer?

6.1 Experimental Methodology

To answer the research questions, we consider randomly selected web pages from a subset of the companies from the Fortune 500 list.[8] We manually identified web pages that use AJAX by browsing the company web sites using the Chrome browser, while enabling

[5] `https://mitmproxy.org/`
[6] `http://www.protractortest.org/`
[7] `https://www.npmjs.com/package/looks-same`
[8] `http://fortune.com/fortune500/`

Table 1: Summary of results.

Company		Tests			Avg. runtime (s)	
Name	Web page	Total	Failures	False positives	Phase 1	Phase 2
1. Amerisource Bergen	Job Openings	25	22	0	33	83
2. Apple	Accessibility	4	0	0	14	57
3. —"—	Buy MacBook	4	2	0	31	82
4. —"—	Customize	4	2	0	18	52
5. —"—	Search Jobs	4	0	0	19	59
6. —"—	Search Support	4	4	0	19	53
7. Bank of America	Search Locations	9	2	0	32	72
8. Berkshire Hathaway	Search Listings	12	2	2	77	162
9. Chevron	Find a Station	30	21	5	34	71
10. Citigroup	News	4	2	0	13	49
11. Exxon Mobil	Job Locations	4	2	0	16	47
12. Fannie Mae	Search	4	0	0	14	42
13. Grainger	Home	4	0	0	36	95
14. McKesson	Home	4	0	0	39	100
15. —"—	Blog Archive	4	2	0	28	108
16. —"—	Event Calendar	4	2	0	21	64
17. —"—	Press Releases	16	9	0	27	82
18. Verizon	Search Locations	4	0	0	31	80
19. Wells Fargo	Home	4	0	0	17	53
20. —"—	Search	4	0	0	17	51
Total		**152**	**72**	**7**		
Average		**7.6**	**3.6**	**0.4**	**25.9**	**68.4**

the "Log XMLHttpRequests" feature and the "Network" panel from the Chrome DevTools,[9] which makes it easy to recognize when an XHR message is being exchanged or an external script is being loaded dynamically. We ignored requests that send analytics data. With this approach, we obtained 20 web pages from 12 different companies, as shown in the "Company" columns of Table 1.

For each of the web pages, we manually create a short user event sequence that exercises some of the dynamic behavior on the web page. Each user event sequence consists of two to nine user events and has been made without any knowledge of the JavaScript code on the web page or the client-server communication. We then carry out the following experiments, on an Ubuntu 15.10 desktop machine with an Intel Core i7-3770 CPU and 16 GB RAM.

Experiment 1. We run AjaxRacer on each subject application using the given manual event sequence. To answer RQ1, we inspect the AJAX event race errors that it reports, and manually check whether each of them can be reproduced. To answer RQ2, we present patterns that we observe in the reported AJAX races. We answer RQ3 by reporting on our experiences during this study with the asynchronous code and the generated reports.

Experiment 2. To answer RQ4, we measure the time needed by AjaxRacer's two phases. For Phase 1, we separately report the time spent on loading the web page and on generating traces for the user events. For Phase 2, we separately measure the time spent on test planning (Algorithm 1) and test execution (Algorithm 2). We repeat the experiments three times and report the average and worst-case running times.

Experiment 3. We run EventRacer on the subject applications using the manually created user event sequences, and answer RQ5 by investigating the results. EventRacer also detects races during the loading of a web page. To estimate how many races arise from the execution of the user event sequence, we analyze the results of EventRacer when no user events are triggered. We report average numbers across three runs. Regrettably, we could not compare to RClassify [25], as it was not available to us.

[9] https://developer.chrome.com/devtools

6.2 Results and Discussion

In this section, we present the results of our experiments, summarized in Table 1, and elaborate on more interesting findings, while addressing RQ1–RQ5.

6.2.1 Effectiveness (RQ1). After Phase 1, AjaxRacer created a total of 152 tests for the web pages in Table 1, which follows from column "Tests". Of the 152 tests, four proved to be infeasible (i.e., one of the user events in these tests was not enabled by the time it was scheduled to be executed). The number of test failures is reported in column "Failures". Each failure reveals a situation where adverse mode execution of a pair of user events leads to a state that is observably different from the corresponding synchronous mode execution. In total, 72 tests failed. The page from Amerisource Bergen produced the highest number of failing tests with 22 failures (row 1). After manually inspecting the results, we found that only seven of the 72 test failures were false positives (column "False positives"). This is a significantly smaller false positive rate than that of existing predictive race detectors such as EventRacer [21]. In particular, each of the 80 succeeding test cases indicates a situation where EventRacer would report a race warning, but where the race is not observable, because ad-hoc synchronization prevents the harmful effects, or the two events from the race commute (i.e., the events have the same effects, irrespective of their arrival order).

Overall, our results show that AjaxRacer is capable of detecting observable AJAX races in real-world web applications with only few spurious warnings.

The fact that the web applications of some of the largest companies in the United States suffer from observable AJAX races demonstrates that this is a widespread problem. Left undetected, they may render the application in an inconsistent state (as we give examples of later in this section). As such, they can frustrate end users and negatively impact their experience. AjaxRacer unveils such situations semi-automatically, with relatively few tests per web page. *In summary, AjaxRacer generated an average of eight test cases per web page, of which half exposed an observable AJAX race.*

False positives. As mentioned above, we observed only seven spurious warnings among the 72 failing tests. Five false positive arose for a web page from Chevron (row 9) because live traffic, which was changing during the execution of the tests (row 9), was being shown on a map. AjaxRacer also reported two false positives for a web page from Berkshire Hathaway (row 8), where the user can search for real estate listings. One test was failing because the screenshot from synchronous mode showed "35,537 Results," whereas the one from adverse mode showed "35,536 Results." Presumably, a listing was removed from the website during the execution of the test. The other false positive from Berkshire Hathaway was similar. We confirmed this behavior by rerunning the tests, which lead to successful executions. If AjaxRacer did not ignore pixels that were already different by the time the web page had been loaded (Section 5), then 11 additional false positives would have been reported (for rows 5, 9, 13 and 19). Generally, there may be other sources

of nondeterminism in the application's UI, such as multimedia resources and third-party entities (e.g., videos and advertisements). Using AjaxRacer, we were able to detect these inconsistencies at a glance, without any need for manual analysis of the code, with the help of the generated reports.

Observable AJAX races are neither the only type of races that exist in web applications, nor did we attempt to reveal all such races within each application. However, the prevalence of observably harmful AJAX races in our subject applications indicates the need for systematic analysis of race-prone code. We used AjaxRacer's reports to gain further insight into the behavior of such code, by manually examining successful tests. As expected, a group of tests succeeded because they lead to the same DOM state, regardless of the ordering of the AJAX events. More interestingly, we encountered another group of successful tests that did not show symptoms of AJAX races, contrary to our initial assumptions (e.g., rows 2, 5, and 8). After a thorough examination of the source code of these applications, we found that the developers had deployed means of remedying the AJAX races. Their strategies not only strengthened our motivation regarding the problematic nature of such races in practice, but also provided insights on common practices for preventing AJAX races (discussed more thoroughly in Section 6.2.6). We also observed AJAX race errors that lead to a series of uncaught exceptions (row 18). AjaxRacer conservatively classified these errors as benign, since they had no observable effects on the screen. Overall, we encountered no cases where the user event sequence lead to an observable AJAX race that was missed by AjaxRacer.

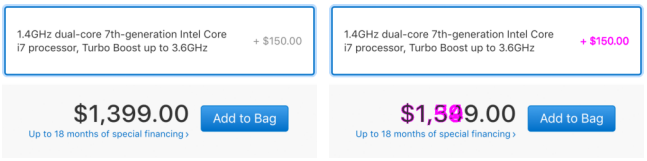
Only four tests among all 152 were deemed infeasible during Phase 2 of AjaxRacer. These tests all belonged to the Press Releases web page of McKesson (row 17). A "next" button on the page was removed from the UI when the number of results did not exceed one page. Therefore, a click event could not be issued on this button if an event that filters the press releases had already been triggered (e.g., an event that clicks on the "November 2017" button).

6.2.2 Race Characteristics (RQ2). We observed that most of the detected AJAX races fit in one of the following categories.

Dataset Queries. Many applications present some data from a database to the user through a list or a table (e.g., web shops). With the overwhelming amount of information available to users, many modern web applications provide means of filtering the displayed dataset, based on users' needs. This is the case for the web pages in rows 1–6, 8, 10, and 17–18 of Table 1. However, when users send multiple queries and the corresponding responses arrive asynchronously, race conditions arise. Such races may cause the displayed data to be inconsistent with the user queries.

Interactive Maps. Many web applications display interactive maps that are used for various purposes, e.g., specifying the locations of retail stores (row 9) or available job postings (row 11). Triggered by user queries, the information overlaid on the maps is updated using AJAX. Conflicts between the user events, discovered by AjaxRacer, lead to incorrect data on the maps of these applications.

Autocompletion. Autocompletion is a feature for generating textual suggestions as soon as a user starts typing in a text field. Such suggestions are often updated asynchronously, which can cause AJAX races that lead to incorrect recommendations. Rows 12–14 and 19–20 correspond to autocompletion features.



(a) Adverse mode. (b) Autogenerated diff.

Figure 5: Inconsistent state when customizing a MacBook.

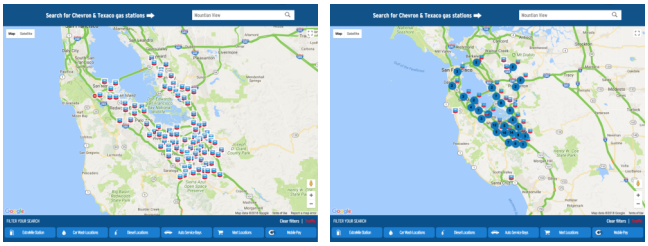
6.2.3 Usefulness (RQ3). Manual analysis of the dynamic, asynchronous, and event-driven behavior of JavaScript applications is a challenging endeavor. To assist developers with understanding AJAX races and locating their root causes, AjaxRacer creates a comprehensive web-based report as the final step. From the report, developers can view the event graphs corresponding to the user events that have been triggered during Phase 1, and see which user events are potentially AJAX conflicting. The report also allows developers to navigate the test results, examine the screenshots that have been taken at different steps during the adverse and synchronous mode executions, and compare the final screenshots.

Example (Apple). On a web page from Apple (row 4), users can customize a MacBook before purchasing it. When the user selects one of the available processors, the UI is updated asynchronously, in response to an AJAX request that fetches the model's information from a web service. AjaxRacer automatically found an AJAX race error on this web page. One of the generated tests is for a user event sequence that clicks on the button for the 1.3GHz processor, and then on the 1.4GHz processor button. Figure 5a shows the undesirable outcome that results from executing this scenario in adverse mode. As it follows from the screenshot, the UI showed that the price would increase by $150 if the user selected the 1.4GHz processor, although it was already chosen (indicated by the blue border). In the screenshot from synchronous mode, a price difference was only shown for the processors that were not selected. The total price of the model was also incorrect in adverse mode. It reflected the price of the previously selected configuration ($1,549.00), rather than that of the 1.4GHz model. Figure 5b shows the diff image that was automatically generated by AjaxRacer. From this image, it was effortless to recognize the consequences of the race, which could otherwise be labor intensive. □

Further analysis of all the generated reports revealed that AJAX event races were common in most subject applications. We examined each report to locate and understand the underlying mechanisms that enabled the races.

Example (Chevron). Recall the motivating example from Chevron, discussed in Section 3. Figure 6 shows the two screenshots that were captured from a test that clicks twice on the "Car Wash Locations" button. The map that results from adverse mode (Figure 6b) did not show any gas stations without a car wash, although it should (the "Car Wash Locations" button is a toggle switch, initially turned off). When the same user event sequence was executed in synchronous mode, the web application correctly showed all the gas stations in the given area, regardless of whether they had a car wash. □

Overall, we found that AjaxRacer's reports provided informative explanations of the causes and effects of each AJAX race.



(a) Synchronous mode. (b) Adverse mode.

Figure 6: Nondeterministic search results on Chevron.

6.2.4 Performance (RQ4). The worst case execution time of AjaxRacer is shown in the rightmost columns of Table 1. In Phase 1, on average, AjaxRacer spent 18 seconds waiting for the web application to load, and eight seconds executing the input user event sequence, while monitoring the execution to build a trace. In the worst case, Phase 1 took 77 seconds (row 8). The test planning (Algorithm 1) took 0.2 seconds in the worst case, including the time required for constructing the event graphs from the traces. In Phase 2, AjaxRacer executed Algorithm 2 for each planned test. On average, this took 69 seconds per test—with 36 seconds spent waiting for the web application to load, and five seconds spent on generating the report. The average running time of Phase 2 was approximately nine minutes, when run sequentially, with a worst case of 34 minutes. However, all tests could easily be executed in parallel. Column "Phase 2" depicts the worst case running time for Algorithm 2, which reflects the time "Phase 2" would take if all tests were executed in parallel. The time required for executing a test in the worst case was below three minutes (row 8). These results demonstrate that the overall performance of AjaxRacer is acceptable for practical use.

6.2.5 Comparison with state-of-the-art (RQ5). When running EventRacer on the subject applications, we found that it reports an overwhelming number of races. As an example, we applied EventRacer to the web application of Berkshire Hathaway (row 8) with a user event sequence that searches for real estate listings, by clicking on the buttons "4+ Beds" and "12+ Beds" (a subsequence of the one given to AjaxRacer). On average, across three runs, EventRacer reported 103,166 races on 37,697 memory locations. 741 of the races were uncovered.[10] The reports contain no information about the effects of the races. When no user events were triggered, EventRacer reported 45,161 races on 28,956 memory locations. This time, 368 of the races were uncovered. Thus, somewhat surprisingly, the two user events caused the number of reported races to approximately double. This shows that EventRacer would still report an overwhelming number of races, even if it had a mechanism for ignoring races that manifest during the loading of web pages. Inevitably, the majority of these races are harmless. Even after manual investigation of the web page, we were unable to detect any observable races. We did find examples of ad-hoc synchronization in the web page, as described below.

[10]Intuitively, a race is *uncovered* if it is guaranteed that no ad-hoc synchronization prevents the two events of the race from being reordered (assuming the happens-before relation is complete).

6.2.6 Common Development Practices. We encountered several practices in the subject applications, which prevented AJAX race errors from manifesting. A simple solution is to avoid the use of AJAX altogether, by reloading the entire web page upon a user event. Although offering a less smooth user experience, this approach was still widespread in practice. Among the applications that utilized AJAX, it was common practice to circumvent AJAX races by disabling UI elements while waiting for a pending response. For example, many applications render a dialog showing a spinner when an AJAX request is sent, until the corresponding response arrives, in a manner that prevents the user from interacting with the web page. While generally offering a better user experience, this approach reduces the responsiveness of the application. Another group of applications used ad-hoc synchronization in a way that did not prevent the user from interacting with the page. For example, on McKesson, an autocompletion feature was implemented in a way that ignored all AJAX response events except the one corresponding to the last request, as documented in the code:[11]

```
40 success: function (data) { // make sure it's the latest request
41   if (__global_counter[container.index] ===
42       requestcounter[container.index]) {
43     ... o.render(container, data, query); ...
```

The following code from Berkshire Hathaway illustrates one of the more sophisticated remedies we found, in terms of the logic and the quality of the user experience.[12]

```
44 var jqXHRs = {};
45 $(checkbox).change(submit);
46 function submit() {
47   if (jqXHRs.search) {
48     jqXHRs.search.abort();
49   }
50   jqXHRs.search = $.ajax(...);
51 }
52 jQuery.noop = function() {};
53 jQuery.ajax = function () {
54   var xhr = new XMLHttpRequest();
55   var jqXHR = { ...
56     abort: function () {
57       xhr.onreadystatechange =
58         jQuery.noop;
59       xhr.abort();
60     } ...
```

When the user clicks on a button labeled "2+ Beds", the function `submit` executes (lines 46–51). This function contacts a web service and updates the search results (line 50). If an AJAX request is already active, the function cancels it by calling the function in lines 56–60, from jQuery 1.7.2. This is done by replacing the `readystatechange` event handler with the empty function in line 52, and calling the native method `abort` of the XHR object.

These countermeasures are helpful in their scope and prevent many AJAX race errors in practice. The mere existence of such treatments indicate that AJAX races are real problems, and that professional developers make an effort to prevent them.

6.3 Threats to Validity

We addressed the external threats of representativeness of our subjects and generality of the investigated scenarios by testing executable sequences of events within widely-used pages of large companies. An internal threat arises from selection of pages, particularly triggering AJAX races as targeted in the scope of this work, a subset of all potential races. To mitigate this bias, we devised

[11]http://www.mckesson.com/js/min/adobe.target.targetcomplete.min.js

[12]The code has been simplified for presentation. It originates from the scripts f345a312-25b7-4242-8165-6dfc8ce834fa and 91ff98c9-a847-4d7b-8bf0-ef5c9697c8ba from http://www.bhhsneprime.com/jscss/23.0.1474/js/.

scenarios for analysis prior to experiment, similar to exploratory testing. Inspection of races and their severity was performed manually, and was thus labor-intensive, and prone to examiners' bias and errors. We alleviated this bias by having two of the authors carefully examining the code and the reports independently.

7 RELATED WORK

It has long been known that JavaScript applications may experience nondeterministic failures depending on the order in which event handlers execute. Steen [23] observed situations where web applications that rely on `setTimeout` to modify a page's DOM representation fail in mysterious ways when browsers parse web pages too quickly or too slowly. Ide et al. [13] point out that these problems can be viewed as a type of race condition, similar to data races in programming languages with concurrency (see, e.g., [5, 8, 9]). One scenario discussed by Ide et al. involves erroneous UI updates that occur when AJAX requests are processed out of order, similar to scenarios we consider. The *throttling* feature in Google's Chrome Developer Tools [15] can be viewed as a poor man's race detector: by simulating various network conditions, situations can be identified where event race errors cause nondeterministic failures.

Zheng et al. [26] present an approach based on static analysis for automatically detecting bugs in web applications where an asynchronous event handler writes to a global variable v, and a user event handler reads v. In such cases, serious errors (e.g., deleting the wrong file on a server) may occur if other event handlers are interleaved that also write to v. Some of the asynchronous scenarios studied in our work have similar characteristics.

Petrov et al. [20] define a happens-before relation for commonly used HTML and JavaScript features and a model of logical memory locations on which web applications operate. These concepts form the basis of WebRacer, a dynamic race detector. Raychev et al. [21] propose a notion of race coverage to eliminate false positives that are due to synchronization deliberately introduced by programmers (ad-hoc synchronization). Intuitively, a race a covers a race b iff treating a as synchronization eliminates b as a race. Nevertheless, predictive race detectors such as WebRacer and EventRacer have been found to report an overwhelming number of races, the majority of which are harmless or benign.

Several projects focus on classifying event races as harmful or harmless. Mutlu et al. [18, 19] apply a dataflow analysis to a trace in order to detect situations where executing racing event handlers under different schedules results in different values being written to persistent storage (cookies and local and session storage). WAVE [10] and R^4 [14] explore executions that can be obtained by reordering events in a sequence of events observed in some initial execution. These tools classify a race as harmful if reordering a pair of conflicting events results in a different DOM, heap state, or uncaught exception. RClassify [25] classifies a race reported by EventRacer as harmful or harmless by generating two executions in which the racing events are executed in both orders and determining if the resulting program states differ in important fields of the DOM, heap, or environment variables. Our work differs from these existing approaches by focusing specifically on AJAX races, by providing detailed explanations for reported issues, and by not relying on the modification of a JavaScript engine.

InitRacer [2] detects race errors that commonly arise during page initialization (form-input-overwritten, late event-handler registration, and access-before-definition errors) using adverse and approximate execution. AjaxRacer follows a similar instrumentation-based implementation technique as InitRacer and provides similar, detailed explanations. However, unlike InitRacer, we focus on detecting AJAX-related races that occur *after* page initialization. InitRacer's adverse execution works by injecting new events whereas AjaxRacer instead delays (AJAX response) events.

Several projects focus on repairing event race errors. ARROW [24] performs a static analysis to determine happens-before relationships between page elements and record these in a causal graph. Races are detected by identifying inconsistencies between the causal graph and def-use relationships inferred from source code order, and prevented by adding causal edges that preclude undesired execution orders. EventRaceCommander [1] is an instrumentation-based tool for repairing event race errors that match patterns that reflect undesirable interleavings (e.g., AJAX requests that are processed out of order). EventRaceCommander avoids these errors by dropping or postponing events so that no undesirable patterns can occur. We use the same mechanism to implement adverse execution.

Several projects focus on detecting event races for other programming languages, including Android [4, 11, 12, 16] and C/C++ [22]. While these works are directly inspired by the work on detecting event races in JavaScript applications [21], applications written in these languages do not rely on AJAX, so the techniques explored in our work do not apply there.

Brutschy et al. [6] show how a generalization of the notion of conflict-serializability can be used to detect race errors in applications that use eventually-consistent data stores.

8 CONCLUSION

We have presented a technique for detecting AJAX event race errors in JavaScript web applications, and described its implementation, AjaxRacer. Our technique uses a combination of light-weight dynamic analysis and controlled execution, and identifies pairs of user events that are potentially AJAX conflicting. For each pair, it generates a test that is expected to fail only if the corresponding AJAX race has observable effects on the screen. Unlike previous techniques, AjaxRacer has been designed specifically to detect AJAX races. As a result, AjaxRacer can detect observable AJAX races in real-world web applications with very few false positives.

In an evaluation on 20 widely used web pages, AjaxRacer detects errors in 12 of them. In total, AjaxRacer generates 152 tests of which 65 reveal AJAX race errors and only seven are false positives. We additionally report on the usefulness of AjaxRacer's comprehensive web-based reports, from which it was easy to locate the root cause and effects of AJAX races, although we had no prior experience with the web pages. In summary, our results show that AJAX race errors are commonplace in web applications and that AjaxRacer is an effective tool for detecting them.

ACKNOWLEDGMENTS

This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 647544).

REFERENCES

[1] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. 2017. Repairing Event Race Errors by Controlling Nondeterminism. In *Proc. 39th International Conference on Software Engineering (ICSE)*. 289–299.

[2] Christoffer Quist Adamsen, Anders Møller, and Frank Tip. 2017. Practical Initialization Race Detection for JavaScript Web Applications. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 66:1–66:22.

[3] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. 2011. A Framework for Automated Testing of JavaScript Web Applications. In *Proc. 33rd International Conference on Software Engineering (ICSE)*. 571–580.

[4] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2015. Scalable Race Detection for Android Applications. In *Proc. 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA)*. 332–348.

[5] Chandrasekhar Boyapati and Martin C. Rinard. 2001. A Parameterized Type System for Race-Free Java Programs. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 56–69.

[6] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin T. Vechev. 2017. Serializability for Eventual Consistency: Criterion, Analysis, and Applications. In *Proc. 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 458–472.

[7] Markus Ermuth and Michael Pradel. 2016. Monkey See, Monkey Do: Effective Generation of GUI Tests with Inferred Macro Events. In *Proc. 25th International Symposium on Software Testing and Analysis (ISSTA)*. 82–93.

[8] Cormac Flanagan and Stephen N. Freund. 2000. Type-Based Race Detection for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 219–232.

[9] Cormac Flanagan and Stephen N. Freund. 2013. RedCard: Redundant Check Elimination for Dynamic Race Detectors. In *Proc. ECOOP 2013 - Object-Oriented Programming - 27th European Conference*. 255–280.

[10] Shin Hong, Yongbae Park, and Moonzoo Kim. 2014. Detecting Concurrency Errors in Client-Side JavaScript Web Applications. In *Proc. 7th IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 61–70.

[11] Chun-Hung Hsiao, Cristiano Pereira, Jie Yu, Gilles Pokam, Satish Narayanasamy, Peter M. Chen, Ziyun Kong, and Jason Flinn. 2014. Race Detection for Event-Driven Mobile Applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 326–336.

[12] Yongjian Hu, Iulian Neamtiu, and Arash Alavi. 2016. Automatically Verifying and Reproducing Event-Based Races in Android Apps. In *Proc. 25th International Symposium on Software Testing and Analysis (ISSTA)*. 377–388.

[13] James Ide, Rastislav Bodik, and Doug Kimelman. 2009. Concurrency Concerns in Rich Internet Applications. In *Proc. Workshop on Exploiting Concurrency Efficiently and Correctly*.

[14] Casper Svenning Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin T. Vechev. 2015. Stateless Model Checking of Event-Driven Applications. In *Proc. 30th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 57–73.

[15] Meggin Kearney and Jonathan Garbee. 2018. Optimize Performance Under Varying Network Conditions. https://developers.google.com/web/tools/chrome-devtools/network-performance/network-conditions

[16] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race Detection for Android Applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 316–325.

[17] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. 2012. Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes. *TWEB* 6, 1 (2012), 3:1–3:30.

[18] Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. 2014. I Know It When I See It: Observable Races in JavaScript Applications. In *Proc. Workshop on Dynamic Languages and Applications (Dyla)*. 1:1–1:7.

[19] Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. 2015. Detecting JavaScript Races that Matter. In *Proc. 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 381–392.

[20] Boris Petrov, Martin T. Vechev, Manu Sridharan, and Julian Dolby. 2012. Race Detection for Web Applications. In *Proc. 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 251–262.

[21] Veselin Raychev, Martin T. Vechev, and Manu Sridharan. 2013. Effective Race Detection for Event-Driven Programs. In *Proc. 28th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages, and Applications (OOPSLA)*. 151–166.

[22] Anirudh Santhiar, Shalini Kaleeswaran, and Aditya Kanade. 2016. Efficient Race Detection in the Presence of Programmatic Event Loops. In *Proc. 25th International Symposium on Software Testing and Analysis (ISSTA)*. 366–376.

[23] Hallvord Reiar Michaelsen Steen. 2009. Websites playing timing roulette. https://hallvors.wordpress.com/2009/03/07/websites-playing-timing-roulette/.

[24] Weihang Wang, Yunhui Zheng, Peng Liu, Lei Xu, Xiangyu Zhang, and Patrick Eugster. 2016. ARROW: Automated Repair of Races on Client-Side Web Pages. In *Proc. 25th International Symposium on Software Testing and Analysis (ISSTA)*. 201–212.

[25] Lu Zhang and Chao Wang. 2017. RClassify: Classifying Race Conditions in Web Applications via Deterministic Replay. In *Proc. 39th International Conference on Software Engineering (ICSE)*. 278–288.

[26] Yunhui Zheng, Tao Bao, and Xiangyu Zhang. 2011. Statically Locating Web Application Bugs Caused by Asynchronous Calls. In *Proc. 20th International Conference on World Wide Web (WWW)*. 805–814.