Practical Initialization Race Detection for JavaScript Web Applications

CHRISTOFFER QUIST ADAMSEN, Aarhus University, Denmark
ANDERS MØLLER, Aarhus University, Denmark
FRANK TIP, Northeastern University, USA

Event races are a common source of subtle errors in JavaScript web applications. Several automated tools for detecting event races have been developed, but experiments show that their accuracy is generally quite low. We present a new approach that focuses on three categories of event race errors that often appear during the initialization phase of web applications: form-input-overwritten errors, late-event-handler-registration errors, and access-before-definition errors. The approach is based on a dynamic analysis that uses a combination of adverse and approximate execution. Among the strengths of the approach are that it does not require browser modifications, expensive model checking, or static analysis.

In an evaluation on 100 widely used websites, our tool InitRacer reports 1 085 initialization races, while providing informative explanations of their causes and effects. A manual study of 218 of these reports shows that 111 of them lead to uncaught exceptions and at least 47 indicate errors that affect the functionality of the websites.

CCS Concepts: • **Software and its engineering → Software testing and debugging**;

Additional Key Words and Phrases: event race detection, JavaScript, dynamic analysis

ACM Reference Format:
Christoffer Quist Adamsen, Anders Møller, and Frank Tip. 2017. Practical Initialization Race Detection for JavaScript Web Applications. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 66 (October 2017), 22 pages.
https://doi.org/10.1145/3133890

1 INTRODUCTION

It is well known that event races are the cause of many errors in JavaScript web applications [Steen 2009]. Such races occur due to nondeterministic ordering of event handlers, for example when program behavior depends on whether a user event appears before or after a script has been loaded. Traditional testing is insufficient for discovering unexpected harmful event orderings, which has motivated the development of a range of powerful techniques and tools to detect event races automatically [Hong et al. 2014; Ide et al. 2009; Jensen et al. 2015; Mutlu et al. 2015; Petrov et al. 2012; Raychev et al. 2013; Wang et al. 2016; Zheng et al. 2011]. However, these existing approaches suffer from various limitations, which makes them unsuitable for production use.

For example, the dynamic race detector EventRacer [Raychev et al. 2013] reports an overwhelming number of races on typical web applications. Most of those races are benign, and it is difficult to classify each race warning as harmful or benign based on the output of the tool [Hong et al.

Authors' email addresses: quist@cs.au.dk, amoeller@cs.au.dk, f.tip@northeastern.edu.

2475-1421/2017/10-ART66
https://doi.org/10.1145/3133890

2014; Jensen et al. 2015; Mutlu et al. 2015]. Many races arise due to ad-hoc synchronization that was added by programmers to *prevent* event race errors, or simply do not affect the application's functionality. The tool by Mutlu et al. [2015] attempts to focus on harmful races, specifically those that affect persistent storage, using a combination of dynamic execution and lightweight static analysis. However, with their technique, any error that may disappear by reloading the web page is considered benign, even though the error may damage functionality or user experience. Additionally, even races that may affect persistent storage are often completely harmless. The *R*^{4} tool [Jensen et al. 2015], which is based on systematic model checking and a notion of approximate replay, is able to produce witness executions that show the consequences of each race, thereby making it easier to determine the degree of harmfulness.

A general limitation of these tools that are based on dynamic analysis is that they can only find race errors in the parts of the code that have been covered by the given user event sequence. Moreover, some races reported by these tools are technically possible but extremely unlikely in practice. As these techniques rely on dynamic analysis using instrumented browsers, we also find that the available prototype implementations quickly fall behind the rapid evolution of browsers and thereby become incapable of processing modern web applications. Other techniques that do not rely on concretely executing the application code but instead apply purely static analysis generally have low precision, caused by the general difficulties in statically analyzing JavaScript code [Andreasen and Møller 2014; Raychev et al. 2013]. Related work, including these techniques, is covered in more detail in Sections 6 and 7.

In this paper, we take a more pragmatic approach towards automated event race detection. We present a tool named InitRacer that has the following properties: (i) it can *detect harmful races with relatively few false positives*[1] compared to the state-of-the-art alternatives, (ii) it is *fast and lightweight* by not requiring expensive model checking or static analysis, (iii) it is *platform-independent*, so it can be implemented without browser modifications, (iv) it is *independent of specific user event sequences* unlike the dynamic race detectors mentioned above, and (v) it produces *informative error messages* to support diagnosing the causes and effects of the races.

We observe that a significant part of the harmful races reported in previous work on event race detection are *initialization races* that occur during the loading and initialization of the web page and do not involve long sequences of user actions, and identify three types of such races: *form input overwritten*, *late event handler registration*, and *access before definition*. We choose to focus entirely on these three common types of initialization races, specifically those that involve at most one user event since the harmful interleavings of such races are more likely to manifest in practice.

Our approach is inspired by two recently proposed techniques for testing Android apps: Thor [Adamsen et al. 2015], which systematically exposes a program to adverse conditions (in our case: events that occur sooner than expected), and AppDoctor [Hu et al. 2014], which uses a notion of approximate execution to speed up testing by directly invoking event handlers instead of faithfully simulating UI events. Our tool InitRacer analyzes a given web page in three phases. In each phase, the page is loaded in a browser via a proxy server that instruments the HTML and JavaScript code to monitor and control execution. In phase 1, the web page is loaded and executed in *observation mode*, without any user events. This allows us to collect information about the behavior of the initialization code in conditions where it is unlikely that serious errors occur, since even simple testing during development would have revealed such errors. In phase 2, the web page is reloaded, this time in *adverse mode* where events are simulated aggressively. Each time an event handler

[1] We call a race a *true positive* if the two possible orderings of the events are feasible. This does not hold for a *false positive*. A race is *harmful* if it is a true positive that leads to an error the developer cares about (note that this determination is subjective). It is *benign* or *harmless* if it is a true positive but does not lead to an error the developer cares about.

http://www.mckesson.com/:

```
1 <input id="search-field" type="text"
2        placeholder="What can we help you find?">
3 <script src="/js/min/search.min.js"></script>
```

http://www.mckesson.com/js/min/search.min.js:

```
4 $(document).ready(function() {
5   var text = getParameterByName("q") || "";
6   $("#search-field").val(text);
7 });
```

Fig. 1. A form-input-overwritten error from http://www.mckesson.com/.

is registered, either by JavaScript code or by HTML code, we eagerly invoke the event handler to mimic a scenario where the event occurs immediately. (Scripts execute non-preemptively, so we naturally wait until the individual scripts have completed before injecting the invocations.) We can thereby detect if, for example, any crashes (i.e., uncaught exceptions) occur when events happen sooner than expected. Finally, in phase 3, the web page is reloaded in *validation mode*. Since phase 2 uses approximate execution rather than simulating high-level user events faithfully, and it injects all possible events in one execution, it is possible that some of the observed crashes are either impossible due to unforeseen interactions between the inserted event handlers or highly unlikely in practice. In validation mode we therefore inject only the crashing event handlers, one at a time. Section 2 gives three examples of different kinds of typical event race errors that can be found with our technique.

In summary, our contributions are as follows.

- We present a light-weight technique for finding initialization race errors in JavaScript web applications. The key idea is to monitor execution of the initialization in *observation mode*, *adverse mode*, and *validation mode*, each providing useful information about the possible behavior of the given application.
- We describe three broad categories of initialization race errors that can be detected using our technique: *form input overwritten*, *late event handler registration*, and *access before definition*.
- The technique is implemented in a tool called InitRacer, and an empirical evaluation shows it to be capable of detecting harmful initialization races in real-world websites. On 100 websites from the Fortune 500 companies, it reports 1 085 event races. Compared to other approaches, it is relatively fast and precise, and it produces informative race error messages that explain the causes and effects of the races. A manual study of 218 of the reports shows that 111 of them lead to uncaught exceptions without directly affecting user experience and that at least 47 indicate errors that affect functionality of the websites.

2 MOTIVATING EXAMPLES

In this section, we discuss three types of initialization race errors that occur commonly, and illustrate them using examples taken from prominent websites.

2.1 A Form-Input-Overwritten Error

A *form-input-overwritten* error manifests when JavaScript code initializes the value of a form input field (e.g., a text field, checkbox, or radio button) *after* the user has already entered a value. Such errors tend to annoy users, because the input they typed is lost and needs to be re-entered.

Consider Figure 1, which shows the relevant fragments of HTML and JavaScript code involved in a real-world form-input-overwritten error that we encountered on http://www.mckesson.com/. In this example, text entered by users in the search field (line 1) may be overwritten by initialization code if page loading is unexpectedly slow. The value of the search field can be modified by the user as soon as the browser has finished rendering it on the screen. However, on line 3, an external script named `search.min.js` is loaded. This script calls jQuery's `ready` function (line 4) to register an event handler for the `DOMContentLoaded` event, which is invoked when the browser has finished parsing the web page. When that happens, the event handler initializes the value of the search field to the result of `getParameterByName("q")` (which retrieves the query parameter `q` from the URL), or the empty string if no query parameter is present (lines 5–6).

We consider the race in Figure 1 to be harmful because the schedule where initialization takes place after the user has entered a value leads to a UI state that is different from the one produced by a schedule where these activities happen in the opposite order. Furthermore, the classification of this event race as being harmful is supported by the following observations: (i) the search field becomes visible to the user by the time it gets declared in the HTML (i.e., no CSS prevents it from being displayed, and its type is not `"hidden"`), (ii) the user can modify the value of the input field (i.e., it is neither `read-only` nor `disabled`), and (iii) there is a potentially long delay (caused by loading the external script) between the time when the search field becomes visible and the time when its value is initialized by JavaScript code.

Our tool InitRacer correctly reports that the value of the `input` field declared at line 79 column 17 in the HTML source code of `http://www.mckesson.com/` is overwritten by the empty string. Additionally, InitRacer identifies the JavaScript operation that overwrites the value of the form field by a stack trace, and takes a screenshot of the web page, where the relevant form field has been highlighted.[2] In contrast, EventRacer [Raychev et al. 2013] does not detect the error (in fact, EventRacer does not detect *any* form-input-overwritten errors). The R^4 tool [Jensen et al. 2015] is based on EventRacer's trace construction and also fails to detect the error in the example. The tool by Mutlu et al. [2015] only considers race errors that affect persistent storage, which is not the case for the error in this example (and for the two following examples).

2.2 A Late-Event-Handler-Registration Error

Another type of initialization race occurs when an event is fired before a corresponding event handler is registered. If this happens, there are two possible outcomes. One possibility is that the event is ignored and nothing happens (e.g., the user clicks on a `button` element before a `click` event handler is installed). In this case, the race is relatively harmless because the user can simply click on the button again to trigger the desired behavior. However, a more serious problem may occur if the event has a default action that needs to be prevented. For example, if an event handler prevents certain characters from being typed into a text field, and the event handler is registered late, then input validation can be bypassed until the event handler is registered.

The code in Figure 2 illustrates such a *late-event-handler-registration* error on `http://apple.com/` where an event handler that prevents the default action can be bypassed by a user event that arrives early. The web page contains a hyperlink "`Search apple.com`" (lines 8–10), and clicking on the link causes a dropdown to appear with a search field. Note that the script `ac-globalnav.built.js` referenced on line 11 is loaded after the search icon has become visible to the user. The "module" with number 194 within this script (lines 22–43) calls the `_initializeSearch` function (lines 26–33). Looking more carefully at the `_initializeSearch` function, we can see that, on line 28, it retrieves a reference to the search link, and then registers `onSearchOpenClick` (lines 34–36) as a `click` event listener on this element (line 19). Hence, from this point onwards, `onSearchOpenClick` is invoked when the user clicks on the search link. When execution of this function reaches line 35, it invokes `l(E)`, which causes `preventDefault` to be called on the event object (line 14). This ensures that the user is not redirected to `http://www.apple.com/us/search`. The event handler then invokes `showSearch` to open the dropdown containing a search field (line 35).

[2] The InitRacer report for this website and the other examples can be found at `http://www.brics.dk/initracer/motivating-examples/`.

http://www.apple.com/:

```
<a class="link-search" href="/us/search">
  Search apple.com
</a>
<script src=".../ac-globalnav.built.js"></script>
```

http://www.apple.com/.../ac-globalnav.built.js:

```
40: [function () {
  d.exports = function b(f) {
    ... f.preventDefault() ...
  }
} , {}],
44: [function () {
  c.exports = function d(i, g, h, f) {
    ... i.addEventListener(g, h, !!f) ...
  }
} , {}],
194: [function (m, d) {
  var i = m("ac-dom-traversal/querySelector");
  var k = m("ac-dom-events/addEventListener");
  var l = m("ac-dom-events/preventDefault");
  v._initializeSearch = function () {
    this.searchOpenTrigger =
      i(".link-search", this.el);
    if (this.searchOpenTrigger) {
      k(this.searchOpenTrigger, "click",
        this.onSearchOpenClick.bind(this));
    }
  };
  v.onSearchOpenClick = function (E) {
    l(E); this.showSearch();
  };
  d.exports = function C() {
    ... this._initializeSearch() ...
  };
}, {
  "ac-dom-events/preventDefault": 40,
  "ac-dom-events/addEventListener": 44,
  "ac-dom-traversal/querySelector": 77 }],
```

Fig. 2. A late-event-handler-registration error from http://www.apple.com/.

At this point, it is clear that the code in Figure 2 exhibits a harmful race because program behavior is quite different depending on whether a user clicks on the search link before or after the event handler has been registered. If the user clicks on the search icon before the event handler has been registered, then the user is redirected to http://www.apple.com/us/search (which is intended for browsers that do not support JavaScript) instead of seeing the dropdown search box. InitRacer correctly identifies this error. It reports that a click event handler, which invokes preventDefault on the event object, is registered too late on the a element at line 146 column 5 in the HTML code. Again, InitRacer reports a stack trace and a screenshot to aid debugging.

Existing tools for event race detection struggle with this example as well. EventRacer finds the race when manually exploring the website, and clicking on the link. However, it does not find the race in auto-exploration mode, even when trying 10 times. On other websites, EventRacer reports false positives and harmless races because it does not take visibility and long delays into account. Furthermore, EventRacer filters away so-called covered races, and, unlike InitRacer, it is limited to detecting errors in code that has been executed by a given user event sequence. EventRacer may also continue to report a warning even after the developers have fixed the late event handler registration.[3] Similarly, R^4 does not find the race when using EventRacer's auto-exploration mode, and it fails to analyze the web page in manual-exploration mode.

2.3 An Access-Before-Definition Error

Sometimes event handlers trigger unexpectedly early, before all application code has been loaded fully. When such event handlers attempt to read a variable, dereference a property of an object, or invoke a function that has not yet been defined, an *access-before-definition* error occurs. Figure 3 shows an example of such an error that we observed on https://www.aetna.com/.

In this code, a menu item "Individuals & Families" is declared (lines 44–49) before a script named s_code.js (line 50). The script declares a top-level variable named s (line 51) and initializes it to the result of invoking the function s_gi. That function assigns a fresh object to a local variable

[3] One approach to fix the error is to register the event handler immediately after the element declaration, which would make the harmful interleaving practically impossible.

https://www.aetna.com/:

```
44 <a href="/individuals-families.html" onclick="
45   Aetna.analytics.omniture.linkCode(
46     'aeCustLnk', 'MMAB:IF', this);
47   s.tl(this, 'o', linkText, null, 'navigate');
48   return false;
49 ">Individuals & Families</a>
50 <script src=".../s_code.js"></script>
```

https://www.aetna.com/.../s_code.js:

```
51 var s = s_gi();
52 ...
53 function s_gi() {
54   var c = "...s.tl=function(o,t,n,vo,f){...}...";
55   var s = new Object();
56   (new Function("s", c))(s);
57   return s;
58 }
```

Fig. 3. An access-before-definition error from http://www.aetna.com/.

s (line 55) and then creates a new function that takes one argument and has the same body as the string stored in the c variable from line 54. Here, the key issue of note is that a function taking five parameters is stored in s.tl. After invoking the newly created function with the object stored in s as argument (line 56), s is returned (line 57).

To understand the problem with this web page, consider what happens if a user clicks on the menu item before the s_code.js script has been loaded. In this scenario, the click event handler of the menu item attempts to invoke s.tl (line 47) before the variable s has been declared. This causes the event handler to crash with an uncaught ReferenceError.

There are several ways a programmer could fix this problem. For example, one could move the loading of the script s_code.js before the declaration of the menu items, so that the user cannot click on the menu items before s.tl is defined. However, this has the unfortunate effect of slowing down the rendering of the web page, because the menu items will not appear until the s_code.js script has been loaded. Alternatively, ad-hoc synchronization could be introduced in the click event handler to account for whether s_code.js has been loaded. In this case, the call can either be skipped, or deferred until s.tl is defined, by setting a timer.

Our InitRacer tool detects the error. EventRacer fails to detect the problem in auto-exploration mode, but does find it in manual-exploration mode. However, after fixing the problem using ad-hoc synchronization, EventRacer *still* reports the same amount of races because the added synchronization code gives rise to an additional (harmless) event race. R^4 does not find the error in auto-exploration mode, and fails to analyze the page in manual-exploration mode.

3 WEB PAGE LOADING IN BROWSERS

Before we can explain how our approach works, we briefly review how browsers load and initialize JavaScript web applications and how event race errors may occur.

Given a URL for a web application, the browser fetches and parses the HTML code while building the corresponding DOM structure. JavaScript code that is embedded within HTML code is executed as it is being encountered during parsing, but it is also possible to load external scripts. Event handlers can be registered either as special attributes in HTML (e.g., onload) or by the JavaScript code (e.g., directly by calling the addEventListener function from the DOM API, or indirectly via jQuery's ready function as in Figure 1). HTML parsing is performed in chunks, which allows for user event processing to be interleaved with parsing. All interleavings are not necessarily possible in practice, though. For example, the browser may choose to parse and render an HTML snippet <div>...</div><div>...</div> in one atomic action, which would prevent the user from interacting with the web page in between the rendering of the two div elements. Scripts may also register *timers*, consisting of JavaScript code that is to be executed after a specified delay. The browser additionally allows event handlers to be associated with different stages of initialization, most importantly the `DOMContentLoaded` event and the `load` event, which signal that the HTML page has been fully parsed and that the page and all sub-resources have finished loading, respectively.

This entire process is single-threaded, so at each point during initialization the browser is either parsing HTML code or executing a script, and each script runs without preemption. However, the scheduling of HTML parsing and script execution is sensitive to the precise timing of events, so interference may occur between individual scripts and between HTML parsing and scripts that access the same JavaScript objects. As a result, execution is nondeterministic, so testing the initialization by a single execution is generally insufficient to cover all possible behaviors—even if we fix all user input, the browser version, the window size, the machine clock, and other factors that may affect the execution.

Our approach is based on a dynamic analysis in which an execution is modeled by a *trace* that consists of different kinds of primitive *actions*. Some of these actions correspond to HTML parsing and others arise from JavaScript execution. Since event handlers execute atomically, we can associate an *event identifier* (EID) with each JavaScript action, identifying the event that triggered the action. As the size of the chunks read by the HTML parser is browser-dependent, we conservatively model the construction of each HTML element as a separate event. The actions are of different kinds:

HTML-element-start[e, o, i] denotes the action of parsing an HTML start tag, where e is a unique EID and o is the constructed DOM element (including its attributes). The argument i specifies extra information about the element: VISIBLE means that the element is currently visible,[4] and for form fields WRITABLE means that the field is neither read-only nor disabled.

focus[e, o] is the action of invoking the `focus` method on a DOM object o, where e is the EID of the current event, or parsing an HTML element o with attribute `autofocus`. In either case, o is focused, meaning that keyboard input is directed to that element.

register-event-handler[e, o, t, h] marks the registration of an event handler h, where e is the EID of the event in which the event handler registration appears, t denotes the event type (e.g., `click`), and o is the DOM or XHR object on which the event handler is registered. This kind of action can appear either due to HTML parsing (e.g., an `onclick` attribute) or due to JavaScript execution (e.g., invoking `addEventListener` or setting `onreadystatechange`).

dispatch[e, h, i] represents the beginning of a script being executed, where e is the EID of the new event and h is the event handler that is about to execute. The argument i specifies whether a “long delay” has happened. We have i = LONG if the event is a timer event with at least 500ms delay,[5] an XHR response event, or an external script loading event.

write-form-field[e, o] means that a script with EID e has written to the value of an HTML form field o.

prevent-default[e] models invocation of `preventDefault` on the event object of the event with EID e, which has the effect that the browser’s default event handling (e.g., following a link, in case of a click on an a element) is disabled for that event.

crash[e] indicates an uncaught exception (such exceptions terminate the current event handler).

loaded represents the pseudo-action that the initialization is completed.[6]

Other actions, in particular JavaScript instructions that read or write other object properties, are abstracted away when forming the trace.

[4] Our implementation uses the *true-visibility* JavaScript library (`https://github.com/UseAllFive/true-visibility`) to determine whether the user can interact with the element.

[5] The 500ms threshold is not significant; changing it to 250ms or 1000ms makes practically no difference for the experimental results presented in Section 6.

[6] Our implementation triggers loaded 5 seconds after the `load` event of the window object, which suffices in practice to await completion of XHR and timers.

In addition to the execution trace, we need the *happens-before* relation $\preceq$ over the EIDs. This relation is easily constructed from the execution, as in previous work [Petrov et al. 2012; Raychev et al. 2013]. Intuitively, $e_1 \preceq e_2$ means that event e_1 must happen before e_2, which is the case if, for example, the trace contains HTML-element-start[e_1,_] before HTML-element-start[e_2,_] or register-event-handler[e_1,_,h] before a corresponding dispatch[e_2,h,_].

Example Loading `http://www.mckesson.com/` (see Figure 1) may yield the trace $\tau_1 \cdot \tau_2 \cdots \tau_6$ where:

$$\begin{aligned}
\tau_1 &= \text{HTML-element-start}[e_1, o_{input}, \text{VISIBLE}, \text{WRITABLE}]\\
\tau_2 &= \text{HTML-element-start}[e_2, o_{script}]\\
\tau_3 &= \text{dispatch}[e_3, h_{script}, \text{LONG}]\\
\tau_4 &= \text{register-event-handler}[e_3, o_{document}, h_{DOMContentLoaded}]\\
\tau_5 &= \text{dispatch}[e_4, h_{DOMContentLoaded}]\\
\tau_6 &= \text{write-form-field}[e_4, o_{input}]
\end{aligned}$$

We have, in particular, $e_1 \preceq e_3$ and $e_3 \preceq e_4$, and there is a long delay at τ_3, which occurs between τ_1 and τ_6. This information suffices to detect the event race error described in Section 2.1, as we shall see in the following section.

4 INITIALIZATION RACE ERRORS

The three examples presented in Section 2 represent different categories of common initialization race errors. We now explain how these categories can be characterized as *trace patterns*, which forms the basis for the design of InitRacer. Such patterns are essentially simple regular expressions over an alphabet of primitive actions.

4.1 Form-Input-Overwritten

Initialization races that lead to form-input-overwritten errors can be characterized using the following trace pattern:

$$\begin{aligned}
\mathbf{P}_1 = {} & \text{HTML-element-start}[e, o, \text{VISIBLE}, \text{WRITABLE}] \cdots\\
& (\text{write-form-field}[e', o] \mid \text{focus}[e', o'])\\
& \text{where } o \text{ is an } \texttt{input} \text{ or } \texttt{select} \text{ element and } o \neq o'
\end{aligned}$$

The pattern $\mathbf{P}_1$ matches any trace that contains HTML-element-start[e, o, VISIBLE, WRITABLE] and this action is followed eventually by either write-form-field[e', o] or focus[e', o'].

If a trace matches HTML-element-start[e, o, VISIBLE, WRITABLE] $\cdots$ write-form-field[e', o] then a script may overwrite the value of the form field o after the user has already changed it. Similarly, if a trace matches the pattern HTML-element-start[e, o, VISIBLE, WRITABLE] $\cdots$ focus[e', o'] where $o \neq o'$ then form field o will lose focus by the time the event with EID e' is dispatched. This is problematic since the user may already be in the middle of modifying the value of the form field o. In both scenarios, the form field must be visible and writable.

The requirement that the form field e in $\mathbf{P}_1$ must be visible by the time it is declared serves to avoid spurious races where the user cannot possibly interact with e before its value changes, or another element receives focus. In the following example, users can only interact with the search field (line 61) by clicking on the button (line 59) after the event listener (line 65), which toggles the visibility of the search field, has been registered.

```
59  <button id="search-btn">Search</button>
60  <div id="dropdown-search" style="display: none">
61    <input type="text" id="search" />
62  </div>
63  <script>
```

```
64 $("#search").val("Enter search terms...");
65 $("#search-btn").click(showSearchDropdown);
66 </script>
```

By that time, however, the value of the search field has already been updated (line 64).

The simple pattern $\mathbf{P}_1$ may, however, lead to harmless races being reported: it may be practically impossible for a user to edit form field o between its creation and the write-form-field or focus action, either because the entire HTML fragment is being parsed in one single chunk, or because the actions happen within a few milliseconds. This is the case for the following example:

```
67 <input id="s" type="text">
68 <script>
69 document.getElementById("s").value = "Enter search terms...";
70 </script>
```

User input to the text field "s" may be overwritten by the script, but only if the edit occurs after line 67 has been parsed but before lines 68–70 have been processed, which is very unlikely. For this reason, we adjust the pattern slightly:

$$\begin{aligned}\mathbf{P}'_{\mathit{1}} = {} & \mathsf{HTML\text{-}element\text{-}start}[e, o, \mathsf{VISIBLE}, \mathsf{WRITABLE}] \cdots \\ & \mathsf{dispatch}[e', \_, \mathsf{LONG}] \cdots (\mathsf{write\text{-}form\text{-}field}[e'', o] \mid \mathsf{focus}[e'', o']) \\ & \text{where } o \text{ is an } \mathtt{input} \text{ or } \mathtt{select} \text{ element, } o \neq o', \text{ and } e \preceq e' \preceq e''\end{aligned}$$

Now the pattern only matches a trace if there is a long delay ($\mathsf{dispatch}[e', \_, \mathsf{LONG}]$) between events e and e''. We use happens-before ($e \preceq e' \preceq e''$) to ensure that only traces are matched where the long delay is guaranteed to occur between the two events and is not just an effect of nondeterministic scheduling. We compare $\mathbf{P}_{\mathit{1}}$ and $\mathbf{P}'_{\mathit{1}}$ empirically in Section 6.

This characterization of form-input-overwritten race errors using trace patterns has several advantages over state-of-the-art alternatives, such as EventRacer and R^4: (i) it identifies form-input-overwritten errors even when no user events are performed, (ii) by taking long delays into account, it avoids reporting interleavings that are unlikely to manifest, and (iii) it does not report spurious races even when the happens-before relation is incomplete, which may happen due to incomplete modeling of the DOM API ($\mathbf{P}'_{\mathit{1}}$ only matches a trace if there is a long delay according to happens-before, whereas existing dynamic race detectors report *more* (spurious) races when the happens-before relation is incomplete).

Interestingly, these trace patterns can even detect some race issues that do not involve any JavaScript code, unlike previous race detection tools. In the following example, the form field on line 73 receives focus after `empty.js` has been loaded.

```
71 <input type="text" />
72 <script src="empty.js"></script>
73 <input type="text" autofocus />
```

Since the scheduling of the actions in this example depends on the network, it is possible that the user has already started editing the form field on line 71 when line 73 is processed.

4.2 Late-Event-Handler-Registration

As illustrated in Section 2.2, undesirable behavior may occur if an event handler is registered too late, i.e., after an event that was supposed to trigger the event handler has already been dispatched. Trace pattern $\mathbf{P}_2$ is designed to identify exactly such situations:

$$\begin{aligned}\mathbf{P}_2 = {} & \mathsf{HTML\text{-}element\text{-}start}[e, o, i] \cdots \mathsf{dispatch}[e', \_, \mathsf{LONG}] \cdots \\ & \mathsf{register\text{-}event\text{-}handler}[e'', o, t, \_] \\ & \text{where } e \preceq e' \preceq e'' \text{ and } \mathit{isUserEvent}(t) \Rightarrow \mathsf{VISIBLE} \in i\end{aligned}$$

Here, the predicate $\mathit{isUserEvent}(t)$ holds if the event type t is a user event (e.g., `click` or `keydown`).

Note that, similar to $\mathbf{P}_1'$ from the previous section, trace pattern $\mathbf{P}_2$ only reports races where the undesirable interleavings are likely to happen in practice due to a long delay between e' and e''. The requirement $\mathit{isUserEvent}(t) \Rightarrow \mathsf{VISIBLE} \in i$ is important for ruling out infeasible interleavings. Indeed, it is not uncommon for user event handlers to be registered on DOM elements that are invisible until the page has been loaded, or until a user event has been performed (recall the example on lines 59–65 from Section 4.1, where the search field was initially hidden). In such situations, the user cannot interact with the element until the event handler has been registered, which existing race detectors such as EventRacer and R^4 do not account for.

Although trace pattern $\mathbf{P}_2$ suffices for identifying late-event-handler-registration errors, it will often lead to an overwhelming amount of reports, since many modern web pages register hundreds of event handlers during loading, many of which are registered late. Yet, most of these web pages work reasonably well. In the following, we therefore refine $\mathbf{P}_2$ to focus on situations that are more likely to affect the user experience.

Late-event-handler-registrations for system events are generally problematic. Web applications often use the `load` event for external scripts, iframes, and images, and if the event handler is registered late, the missing event handler execution can only be remedied if the user reloads the page.

For user events, as discussed in Section 2.2, there are situations where the user simply needs to repeat the user event after a late-event-handler-registration error in order to obtain the desired behavior, which is annoying but not a major problem. It is generally more problematic if the event handler calls `preventDefault` on the event object, since this prevents the browser's default event handling (recall the example from Section 2.2). We therefore refine the pattern using the prevent-default action to match only traces where the user event handler invokes `preventDefault`:

$$\begin{aligned}\mathbf{P}_2' = {}& \mathsf{HTML\text{-}element\text{-}start}[e, o, i] \cdots \mathsf{dispatch}[e', \_, \mathsf{LONG}] \cdots \\ & \mathsf{register\text{-}event\text{-}handler}[e'', o, h] \cdot \tau \\ & \text{where } e \preceq e' \preceq e'' \text{, and } \mathit{isUserEvent}(t) \Rightarrow \big(\mathsf{VISIBLE} \in i \wedge \\ & \quad \tau \text{ matches } \mathsf{dispatch}[e''', h, \_] \cdots \mathsf{prevent\text{-}default}[e''']\big)\end{aligned}$$

The sub-pattern $\mathsf{dispatch}[e''', h, \_] \cdots \mathsf{prevent\text{-}default}[e''']$ applies if t is a user event and checks that the event handler h has invoked `preventDefault` in the trace. Existing race detectors do not take into account whether late-event-handler-registration races affect the browser's default event handling.

4.3 Access-Before-Definition

Access-before-definition errors arise when a variable or object property is read before it has been initialized, as in the example from Section 2.3. In JavaScript, a `ReferenceError` is thrown if an attempt is made to read a variable that has not been declared, and a `TypeError` is thrown when dereferencing a property from `null` or `undefined`, and when invoking a non-function value. As mentioned in Section 3, such exceptions cause the current event handler to abort, which may leave the program in an undesired state. Similar to late-event-handler-registration errors, crashes in system event handlers are generally more problematic than ones in user event handlers.

For simplicity, we classify access-before-definition errors as being harmful only if they lead to uncaught exceptions.[7] Uncaught exceptions may of course appear without relation to initialization races. Such exceptions are likely benign, because they also would manifest during ordinary testing.

[7] For future work, it may be interesting to also consider the side-effects of code that was not executed because of a crash, similar to the use of prevent-default in Section 4.2.

For this reason, we want to identify each event handler that may crash *during* initialization but not *after* initialization.

The following trace pattern $\mathbf{P}_{3a}$ matches a trace τ_a if it contains an event handler h that crashes during initialization, i.e., before the loaded action:

$$\begin{aligned}\mathbf{P}_{3a} = {} & \text{register-event-handler}[e_a, o_a, t, h] \cdots \\ & \text{dispatch}[e'_a, h] \cdots \text{crash}[e'_a] \cdots \text{loaded}\end{aligned}$$

If another trace τ_b witnesses that h may also crash after initialization, then we filter away the crash in h as likely benign. This situation is captured by the trace pattern $\mathbf{P}_{3b}$:

$$\begin{aligned}\mathbf{P}_{3b} = {} & \text{register-event-handler}[e_b, o_b, t, h] \cdots \\ & \text{loaded} \cdots \text{dispatch}[e'_b, h] \cdots \text{crash}[e'_b]\end{aligned}$$

Section 5 explains how to obtain the traces τ_a and τ_b.

Compared to existing race error detectors, this simple approach has several important properties. First, it does not report an overwhelming number of harmless reports in presence of ad-hoc synchronization, unlike EventRacer. Second, similar to R^4 but unlike EventRacer, it only reports races that actually lead to errors. As we shall see in the next section, we also leverage adverse execution to become independent of specific user event sequences.

5 THE INITRACER APPROACH

InitRacer works in three phases that execute the initialization of the given web application in different modes, controlled by instrumentation performed by a proxy server.

Phase 1: Observation Mode Execution. The first phase is dedicated to detecting form-input-overwritten and late-event-handler-registration errors using trace patterns $\mathbf{P}'_1$ and $\mathbf{P}_2$. Errors characterized by these patterns can be detected by merely observing the instructions that execute during initialization. Thus, in phase 1 InitRacer simply opens the given web page and collects the trace until the loaded action occurs as described in Section 3.

To emit the necessary actions for trace pattern $\mathbf{P}'_1$, the instrumentation intercepts assignments to the `value` property of `input` elements, assignments to the `selectedIndex` property of `select` elements, invocations of the `focus` method of HTML elements, and declarations of HTML elements with the `autofocus` attribute. The relevant actions for $\mathbf{P}_2$ are generated by intercepting invocations of the `addEventListener` method and assignments to event attributes (e.g., `onclick`, `onreadystatechange`). Event handler attributes in the HTML are ignored; such registrations are never late. InitRacer intercepts all property assignments by also dynamically instrumenting code that is passed to `eval` at runtime.[8]

Determining happens-before Patterns $\mathbf{P}'_1$ and $\mathbf{P}_2$ both rely on the happens-before relation. This relation is built on-the-fly by monitoring the parsing of HTML elements and the execution of scripts. As explained in Section 3, each action takes place in the context of an event. An important step in building the happens-before relation is to wrap each function that is registered as an event handler, in order to record the current event e_r at the time of the registration. When the wrapper is eventually invoked, it is then possible to insert a happens-before edge between the event in which the handler was registered and the current event e_n, i.e. $e_r \preceq e_n$.

Note that care needs to be taken when wrapping event handlers because an application may unregister event handlers (using the function `removeEventListener`). If the application passes a reference to the unwrapped event handler function, rather than the wrapped one, the removal

[8] The implementation uses the *falafel* instrumentation library (https://github.com/substack/node-falafel).

fails silently. For this reason, InitRacer maintains a map from functions to their wrappers, and intercepts calls to `removeEventListener` to ensure that the correct function is passed.

Form-input-overwritten detection Trace pattern $\mathbf{P}'_{\mathit{1}}$ for form-input-overwritten errors is susceptible to false positives, as illustrated by the following example:

```
74 <input type="text" id="search" value="Default" />
75 ... // assume long delay
76 <script>
77 var input = document.getElementById('search');
78 if (input.value === 'Default') {
79   input.value = 'Enter search terms...';
80 } else { /* avoid overwriting user input */ }
81 </script>
```

Executing this code produces a trace that matches $\mathbf{P}'_{\mathit{1}}$, but the write-form-field action (line 79) is guarded by line 78, which checks whether the user has changed the field value. To avoid such situations, we intervene in the observation mode execution as follows: (i) when a form field is declared, InitRacer immediately changes its value to a random non-default one, and (ii) when the web page has loaded, InitRacer checks if the value of each form field has changed since its declaration.

Phase 2: Adverse Mode Execution. As discussed, phase 1 provides information for patterns $\mathbf{P}'_{\mathit{1}}$ and $\mathbf{P}_2$. The purpose of phase 2 is to collect information needed for the patterns $\mathbf{P}'_2$ and $\mathbf{P}_{\mathit{3a}}$ (in particular, trace τ_a). Notice that in $\mathbf{P}'_2$, the sub-pattern dispatch$[e''', h, _] \cdots$ prevent-default$[e''']$ will not appear in the trace unless the event handler h has been triggered. Similarly, in $\mathbf{P}_{\mathit{3a}}$ and $\mathbf{P}_{\mathit{3b}}$, the sub-pattern dispatch$[e', h] \cdots$ crash$[e']$ will not be matched unless h has been triggered. In phase 2, InitRacer reloads the given web page in a mode where it systematically simulates events during initialization, in an attempt to reach the relevant prevent-default and crash actions.

Some web pages register hundreds (or even thousands) of event handlers during initialization, so repeatedly reloading the page and injecting a single one of the individual events would not scale well. (Waiting for initialization to finish may take up to 20 seconds, due to the instrumentation needed for the analysis.) We therefore use the idea of *adverse execution* from Thor [Adamsen et al. 2015]: in a single execution, InitRacer simulates *all* events for which event handlers have been registered. The events are injected eagerly, as soon as possible after the event handlers have been registered. This does not necessarily lead to the "most adversarial" event ordering, but one that developers would not normally observe.

Rather than faithfully simulating users moving the mouse pointer over the screen and interacting with the web page via mouse and keyboard clicks, etc., we use the idea of *approximate execution* from AppDoctor [Hu et al. 2014]: to simulate an event InitRacer simply invokes the event handler function directly. This is fast and easy to implement, in particular because it does not require browser modifications. The drawback is that the resulting executions may not be feasible in ordinary execution. For example, we bypass the browser's event bubbling/capturing mechanism, and we ignore the fact that it is unrealistic to trigger `keyup` events without preceding `keydown` events. Phase 3, which we explain later in this section, can filter away some false positives that arise due to artifacts of adverse and approximate execution.

In more detail, phase 2 of InitRacer works as follows. At event handler registrations, InitRacer wraps the event handler function using `try-finally` so that we can inject code at the exit of the event handler. The code we inject generates fake event objects and invokes the event handlers that have been registered. Each invocation is put into a `try-catch` block so that we can detect crash events.[9] Note that, because we invoke the event handlers directly, we do not have to worry about the browser's default actions for the events (e.g., submitting forms, or following links). An alternative approach to inject events would be to use the built-in function `dispatchEvent`, which simulates events more faithfully, but it does not allow us to control exactly which event handlers are executed. Certain kinds of events—`DOMContentLoaded`, `load`, and `unload`—play a special role in the lifecycle of the web page, so InitRacer does not inject invocations of event handlers for those events. Our current implementation also does not inject `readystatechange` events, since it is difficult to automatically generate a meaningful XHR response; in a future version we will record and reuse the responses from phase 1. Finally, executing event handlers may have undesired side effects, such as form submission, page redirects, alert popups, opening the print dialog, etc., so InitRacer intercepts and disables such effects. For example, form submissions are disabled by monkey-patching the `HTMLFormElement.prototype.submit` method.

It is important to keep observation mode and adverse mode apart. Using adverse mode execution as basis for detecting form-input-overwritten errors would result in false positives, caused by injection of event handlers with write-form-field actions. As an example, the event handler for a "reset form" button writes to form fields, but that does not imply existence of a form-input-overwritten error. Also, to limit interference of injected events, which could cause spurious matches of $\mathbf{P}'_2$, register-event-handler actions that belong to the injected events are omitted from the generated traces.

The existing tools EventRacer and R^4 find races only in code that has been executed by a given user event sequence. EventRacer does have an automatic exploration mode, but that is quite limited and only triggers a small fraction of the relevant events. In contrast, InitRacer's adverse mode execution simulates *all* events for which event handlers have been registered, to observe their effect when interleaved in the initialization of the web application.

Phase 3: Validation Mode Execution. As explained above, the aggressive injection of event handler invocations in phase 2 may generate traces that are impossible or unlikely in actual execution, which may result in harmless races but also false positives especially for the access-before-definition error detection. For this reason, phase 3 attempts to validate potential initialization errors by reloading the web page again, once for each potential access-before-definition error that was detected in phase 2, and injecting only the single event handler containing the crash action. If this causes the error to disappear, InitRacer by default treats it as a false positive and omits it from its report.

This approach has another benefit: it eliminates (true) errors that only manifest if multiple user events occur during the initialization. Such errors are inevitably less likely to occur in practice, which is why we aim for detecting initialization race errors that involve at most one user event. Note that InitRacer can be adjusted to explore race errors that only manifest when multiple user events are triggered during the initialization. For example, given an error that manifests in adverse execution mode where all event handlers are executed eagerly, delta debugging [Zeller and Hildebrandt 2002] could be applied to find the minimal set of event handlers that must be triggered during initialization to detect the error.

In addition to injecting a single event handler *during* loading, InitRacer also injects the same event handler *after* the page has loaded, such that the trace (τ_b) can be used for trace pattern $\mathbf{P}_{3b}$. This serves to identify access-before-definition errors that only manifest during initialization. The number of validation mode executions is generally much lower than the total number of events injected in phase 2, so keeping these two phases separate is important for performance.

[9] If one wants to find a broader range of access-before-definition errors beyond uncaught exceptions, it is possible to apply the dynamic analysis from DLint [Gong et al. 2015].

This approach is inspired by Thor that performs a similar validation step to isolate the causes of failures [Adamsen et al. 2015].

The validation mechanism requires a way to identify the same event handler registration across two executions. We employ a pragmatic approach that we have found to work well: event handler registrations are identified by the name of the target (e.g., `div`, `img`, `document`), the target's location in the source code (if available), the target's visibility, the event type (e.g., `click`), and the event handler's source code (found by calling `toString` on the function).

Validation mode execution is not a perfect filter against harmless races and false positives. It eliminates most interference due to injected event handlers in adverse mode, but makes no attempt to prevent false positives that may appear due to approximate execution. This is a pragmatic design choice (implementing a precise validation mechanism, as AppDoctor's *faithful mode* [Hu et al. 2014], is impossible without browser modifications).

Error Diagnosis. To support error reproduction and debugging, each report generated by InitRacer concisely shows the relevant actions, including source code and stack traces, and with screenshots highlighting the involved HTML elements. The reported issues are grouped according to the three categories and the involved actions.

Figure 4 (i) presents the report that has been generated by InitRacer for `http://www.apple.com/`. The bottom of this report shows a screenshot of the web page, where the UI elements that are involved in a race have been highlighted. For example, the magnified part of Figure 4 (i) highlights that InitRacer has reported a late-event-handler-registration warning for the `click` event type (with ID 6) for the search icon. (The error that causes this warning is the one described in detail in Section 2.2.) By inspecting the warning with ID 6 in the table, it can be seen (from column "Name") that the problematic event handler is registered on the a element that is declared on line 146 column 5 in the HTML source code, and (from column "Stack trace") that the event handler registration is performed in the file `ac-globalnav.built.js`. These pieces of information are useful for diagnosing the warning. A natural first step in debugging is to inspect the expected behavior. Figure 4 (ii) shows the screen that results from clicking on the search icon that has been highlighted by InitRacer *after* the web page has finished loading. In this scenario, the user is presented with a search field in a dropdown. Figure 4 (iii), on the other hand, shows the screen that appears when clicking on the search icon *before* the page (and, in particular, `ac-globalnav.built.js`) has loaded. In this case, the user is redirected to `http://www.apple.com/us/search` (see Section 2.2 for details). This behavior can easily be reproduced by simulating a slow network, for example, by enabling throttling (on the "Network" panel) in Chrome DevTools.[10]

6 EVALUATION

We aim to answer the following research questions through an empirical evaluation:

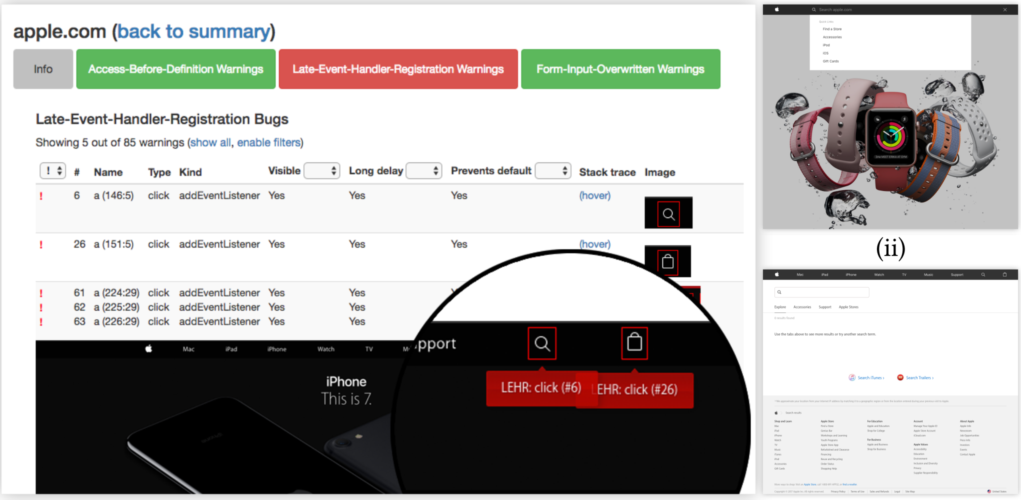
RQ1 How many form-input-overwritten errors, late-event-handler-registration errors, and access-before-definition errors does InitRacer report on real websites?

RQ2 How fast is InitRacer on real websites?

RQ3 How often do the warnings reported by InitRacer identify actual errors?

RQ4 How does InitRacer compare with EventRacer [Raychev et al. 2013] and R^4 [Jensen et al. 2015] in terms of usefulness?

[10] `https://developers.google.com/web/tools/chrome-devtools/`


apple.com (back to summary)
Info
Access-Before-Definition Warnings
Late-Event-Handler-Registration Warnings
Form-Input-Overwritten Warnings
Mac
iPad
iPhone
Watch
TV
pport
iPhone
This is 7.
LEHR: click (#6)
EHR: click (#26)

Late-Event-Handler-Registration Bugs

Showing 5 out of 85 warnings (show all, enable filters)

!	#	Name	Type	Kind	Visible	Long delay	Prevents default	Stack trace	Image
!	6	a (146:5)	click	addEventListener	Yes	Yes	Yes	(hover)	
!	26	a (151:5)	click	addEventListener	Yes	Yes	Yes	(hover)	
!	61	a (224:29)	click	addEventListener	Yes	Yes			
!	62	a (225:29)	click	addEventListener	Yes	Yes			
!	63	a (226:29)	click	addEventListener	Yes	Yes			

(i) (ii) (iii)

Fig. 4. (i) The InitRacer report for `http://www.apple.com/`, where the magnified part shows the warning markers placed by InitRacer. (ii) The screen resulting from clicking on the search icon *after* the page has loaded, which is the normal behavior. (iii) The screen resulting from clicking on the search icon *before* the page has fully loaded.

Experimental Methodology. The empirical evaluation is based on websites of the 100 largest companies from the Fortune 500 list, similar to evaluations of previous event race tools [Adamsen et al. 2017; Jensen et al. 2015; Raychev et al. 2013]. To ensure reproducibility of our results we use an intercepting HTTP proxy[11] for recording the server responses observed in interactions with the web pages under consideration. The implementation of InitRacer, recordings of server responses, and all experimental data are available at `http://www.brics.dk/initracer/`.

Answers to RQ1 and RQ2 are obtained by running InitRacer on each of the web pages and counting the number of reported issues in each category. Given the complexity of the websites under consideration, only a representative subset of these issues is considered to answer RQ3 and RQ4. We manually attempt to reproduce all form-input-overwritten errors. For late-event-handler-registration and access-before-definition errors, we consider all warnings from 10 randomly selected websites. All experiments are run in Google Chrome on Ubuntu 15.10 (Intel Core i7-3770 CPU and 16 GB RAM).

RQ1: Number of Reported Warnings. Table 1 shows the number of warnings generated by InitRacer for the front pages of the 100 websites, in each of the three categories. The columns in this table show, for each website,[12] the number of form-input-overwritten (FIO) warnings, late-event-handler-registration (LEHR) warnings, and access-before-definition (ABD) warnings. For LEHR and ABD, we further distinguish between user and system event handler registrations (cf. Sections 4.2–4.3). We next discuss the findings for each category.

[11] `https://mitmproxy.org/`

[12] InitRacer does not report any warnings for 30 of the 100 websites; those have been excluded from the table.

Table 1. Results.

	FIO	LEHR		ABD	
Website		User	Sys.	User	Sys.
21cf.com	0	0	0	3	0
3m.com	0	7	0	0	0
aa.com	0	19	0	2	0
adm.com	0	0	0	13	0
aetna.com	0	13	0	32	3
aig.com	0	1	0	0	0
allstate.com	0	4	0	0	0
amazon.com	0	5	1	0	0
americanexpress.com	0	12	0	1	0
anthem.com	0	1	0	1	0
apple.com	0	5	0	0	0
att.com	0	0	20	0	0
bankofamerica.com	4	15	0	1	0
bestbuy.com	1	4	0	1	0
cardinalhealth.com	0	7	0	0	0
chevron.com	0	8	0	0	0
chsinc.com	0	7	0	22	0
cigna.com	0	24	0	3	0
cisco.com	0	2	0	1	0
conocophillips.com	0	0	0	1	0
costco.com	0	2	0	3	0
cvshealth.com	0	8	0	5	0
deere.com	0	6	0	80	0
delta.com	2	10	0	12	0
directv.com	1	1	2	4	0
disney.com	0	1	0	0	0
dupont.com	0	3	0	0	0
energytransfer.com	0	0	0	1	0
express-scripts.com	0	21	0	0	0
fedex.com	0	0	0	8	0
freddiemac.com	0	5	0	0	3
ge.com	1	0	0	0	0
gm.com	0	41	0	0	0
goldmansachs.com	0	0	0	1	0
halliburton.com	0	12	0	9	0
homedepot.com	0	1	0	0	0
honeywell.com	0	1	0	1	0
hp.com	0	7	0	0	0
humana.com	0	11	0	11	1
ibm.com	0	1	0	0	0
ingrammicro.com	0	0	0	2	0
intel.com	1	0	0	1	0
intlfcstone.com	0	25	0	1	0
jnj.com	0	1	0	0	0
lockheedmartin.com	0	0	0	1	0
lowes.com	1	2	0	0	0
mckesson.com	1	0	0	0	0
merck.com	0	2	0	3	0
metlife.com	0	12	4	0	0
microsoft.com	0	12	0	0	0
mondelezinternational.com	0	0	0	1	0
morganstanley.com	0	51	0	0	0
nationwide.com	0	1	0	11	0
pepsico.com	0	3	0	1	0
pfizer.com	0	1	0	1	0
phillips66.com	0	0	0	1	0
prudential.com	1	1	0	0	0
statefarm.com	0	11	0	0	0
tsocorp.com	0	3	0	0	0
united.com	10	16	0	0	0
unitedhealthgroup.com	0	1	0	31	0
ups.com	0	0	0	7	3
us.coca-cola.com	0	2	0	1	0
utc.com	0	8	0	1	0
verizonwireless.com	0	24	0	17	0
walgreens.com	0	0	10	2	0
walmart.com	0	41	134	1	0
wellsfargo.com	0	6	0	1	0
wfscorp.com	0	5	1	0	0
xfinity.com	1	84	0	7	0
total	24	577	172	308	10

FIO As can be seen in the 'total' row of the table, InitRacer finds a total of 24 FIO warnings due to matches with pattern $\mathbf{P}'_1$.[13] Without the check for form field value changes, 2 additional warnings would have been reported. 10 of the 24 warnings are due to write-form-field and 14 are due to focus.

LEHR InitRacer reports a total of 749 LEHR warnings due to matches with pattern $\mathbf{P}'_2$,[14] of which 577 involve user event handlers, and 172 involve system event handlers.

[13] The less precise race pattern $\mathbf{P}_1$, reports 4 additional write-form-field warnings, suggesting that most issues do involve a long delay. The long delay requirement in $\mathbf{P}'_1$ is still useful because, without it, a race detector may report a warning even *after* the error has been fixed.

[14] Using the less precise pattern $\mathbf{P}_2$, which does not take into account long delays or whether a registered user event handler calls preventDefault, leads to an overwhelming increase in the number of warnings: 9 189 and 585 for user and system event handler registrations, respectively.

ABD The 100 τ_a-traces collected by InitRacer gives rise to 318 matches with $\mathbf{P}_{3a}$ that can be validated by injecting only a single event handler. Of these, 308 involve user event handlers and 10 involve system event handlers. Without the validation phase, 23 additional warnings (possibly due to interference) would have been reported. Of the 318 validated warnings, 246 user event handlers only crash during initialization (i.e., the corresponding trace τ_b from validation mode does not match $\mathbf{P}_{3b}$). InitRacer was only able to confirm that 1 of the 10 system event handlers crashes after initialization. The fact that InitRacer succeeds in validating most warnings suggests that adverse mode execution does not introduce too much interference. In fact, on 39 of the 47 websites that have at least one ABD crash, InitRacer is able to validate *all* crashing user and system event handlers.

RQ2: Performance. In InitRacer's observation phase and adverse phase, the instrumented web page is loaded just once. Although this is significantly slower compared to loading the original web page (mostly due to instrumentation overhead and taking screenshots) this can be done in 7 to 40 seconds depending on the web page (21 seconds on average).

In the validation phase, InitRacer loads the web page once for each ABD warning. On average, this takes around 1 minutes per website with at least one ABD warning (23 minutes for `https://www.deere.com/`, which has the most ABD warnings), while it is free for the remaining 53 websites.

RQ3: Qualitative Study. **FIO** We manually inspected all 10 FIO warnings where the user's input to a text field is overwritten. For 6 of the warnings we were able to reproduce the errors. 2 of the warnings are spurious due to false positives from the *true-visibility* JavaScript library that is used by InitRacer, and can be fixed by better visibility checking.[15] We were unable to reproduce the remaining 2 warnings in Google Chrome (apparently due the browser's chunk size; in both cases, the form field did not become visible on the screen until after the field's value had already been updated).

LEHR We manually investigated all 75 LEHR warnings from 10 randomly selected websites. On 8 websites InitRacer reports 27 warnings associated with functionality that is either disabled or malfunctioning during initialization. The most commonly occurring situation is that functionality is disabled during loading. For example, InitRacer finds hyperlinks that fail to open a dialog, a menu, a login form, or redirect the user to a different page, as well as features such as auto-completion that are not enabled during loading. On `http://www.bestbuy.com/`, InitRacer detects a scenario where the web page redirects the user when signing up for a newsletter instead of sending an XHR request, as the web page does after loading. As with the example from Section 2.2, the user's experience is effectively degraded to the one offered to users whose browsers do not execute JavaScript. In another example from `https://www.aetna.com/` a late-event-handler-registration for the `submit` event causes the user's search query to be lost upon submitting a form.

The remaining 48 LEHR warnings are spurious. Of these, 11 are false positives from *true-visibility* and 1 could not be reproduced (possibly due to the browser's chunk size). Another 5 (all from `https://www.aetna.com/`) are due to `click` event handlers registered on hyperlinks that invoke `preventDefault` on the event object and then redirects the user by assigning `window.location`. These event handlers are superfluous: they re-implement the browser's mechanism for redirecting the user when a hyperlink is clicked, and one could argue that these should simply be removed. The remaining 31 of the 48 spurious warnings are due to event handlers that track the user once

[15] Although most false positives originating from *true-visibility* can be fixed easily, some are non-trivial. For example, a form field from `https://www.bankofamerica.com/` takes up 230x32 pixels on the screen, but is visually indistinguishable from the background until the page has loaded.

the page has been loaded. For example, 12 are from https://www.microsoft.com/, where an event handler is registered that prevents the browser from redirecting the user, such that an XHR request can be sent before the redirection.

ABD On 10 randomly selected web pages InitRacer reports 133 crashes, of which we are able to reproduce 125 manually. Of these 125 errors, 14 affect user experience. In an example from https://www.ups.com/, an uncaught exception in a `click` event handler causes the user to get redirected to the splash page when clicking "Change Language", rather than being presented with a dropdown (as after the page has loaded). The remaining 111 errors are caused by calls to, e.g., analytics libraries that do not get loaded until the end of the initialization. On https://www.deere.com/, which has 79 such crashes, the developers are aware of the problem and test if the library has been loaded:

```
if (omniEvents) omniEvents.globalNav(this, 'header:Products');
```

Unfortunately, the test itself leads to a `ReferenceError`.

The 8 warnings that are not reproducible involve interleavings that are extremely unlikely. Each case involves an event handler that crashes by the time it is registered (due to invoking a function that has not yet been declared), but only until the browser has parsed a few more HTML elements.

Debugging Although InitRacer reports relatively many LEHR and ABD warnings for some websites, many of those warnings have similar characteristics, and InitRacer's screenshots and grouping of related issues reduces the debugging effort significantly. For example, the 5 harmless LEHR warnings from https://www.aetna.com/ are due to 5 menu items on the page that share the same event handler. InitRacer groups these races, making it easy to recognize that they are related, and the provided screenshot reveals that each warning is associated with a corresponding menu item, making it trivial to determine that they are in fact all caused by the same problem. Overall, only 11 of the 48 spurious LEHR warnings needed to be investigated in detail.

Furthermore, the three error categories have the desirable property that warnings can easily be tested for reproducibility. For example, a FIO error can be tested for reproducibility by attempting to modify the value of the involved form field before the JavaScript instruction identified by InitRacer updates the form field's value. For the FIO error from http://www.mckesson.com/ (Section 2.1), this can be done by postponing the script `search.min.js`. To carry out this evaluation, we used the mitmproxy tool to restrict a URL of our choice to schedules that are unlikely in normal circumstances.

InitRacer also makes it easy to debug the 111 analytics related ABD warnings. For example, the report indicates that the 79 warnings from https://www.deere.com/ are all due to a `ReferenceError` on the variable `omniEvents`, and the screenshot clearly indicates which HTML elements are involved.

In summary, the reports generated by InitRacer were sufficiently informative to enable us to quickly identify and reproduce numerous initialization race errors in (often obfuscated) websites that we were not previously familiar with. We generally needed only few minutes to inspect and reproduce a single race, and all 218 races were classified in approximately one day of work.

RQ4: Comparison to State-of-the-Art. EventRacer often reports an overwhelming amount of event races. On https://www.united.com/ alone, EventRacer reports 16 822 races, of which 533 are uncovered.[16] After manually applying EventRacer to 10 initialization errors detected by InitRacer, we found that only 1 of the 10 is uncovered. When applying EventRacer to all 100 websites in our study, we found that it reports more than 1 000 races for 49 of the websites, and more than 10 000 races for 19 of them. Inevitably, most are either harmless or false positives.

[16] In an *uncovered* race, both execution orderings of the corresponding memory accesses are guaranteed to be possible.

The reports generated by EventRacer do not support debugging very well. An EventRacer report is simply a trace of low-level read and write operations, structured in events, with the additional information that two events in the trace are unordered according to the happens-before relation.

R^4 systematically explores the possible schedules. An R^4 report consists of a specific sequence of steps taken by the browser in order to expose a given error, e.g., an uncaught exception or visual difference. It is very difficult to reproduce errors detected by R^4, since there is no means to replay a specific schedule in a real browser and no support is offered for further diagnosis. Each warning reported by InitRacer can be tested for reproducibility in a well-defined way.

Both EventRacer and R^4 build on an old version of WebKit, which makes these tools platform-dependent and also leads to problems when analyzing modern web pages. For example, the message "Your browser is not supported" is shown when loading `http://www.ford.com/` and 13 other of the 100 websites from our study. On `https://www.kroger.com/`, the web page remains blank even after loading. On `http://www.citigroup.com/` and `http://www.marathonpetroleum.com/`, EventRacer fails to analyze the web page of interest, since its auto-exploration triggers a redirect to another page, and EventRacer, unlike InitRacer, does not disable such undesirable side effects (Section 5). Similarly, `http://www.ge.com/` keeps reloading in an infinite loop when analyzed using EventRacer. Updating EventRacer and R^4 is a nontrivial task; the tools are more than 80 000 commits behind the latest version of WebKit.[17]

Threats to Validity. It is possible that the websites considered in our evaluation do not provide a representative mix of programming styles and JavaScript feature usage. However, this style of evaluation was also used in previous work on event race tools [Adamsen et al. 2017; Jensen et al. 2015; Raychev et al. 2013]. A related issue is that the websites under consideration evolve continually. To enable reproducibility, we recorded all websites, and will make InitRacer and recorded websites available as an artifact. Full reproducibility is not always possible. For example, if a website exhibits nondeterminism that is unrelated to user events, then executions may differ from ones we observed. To the best of our knowledge such situations do not affect the conclusion of our experiments. Furthermore, while InitRacer is platform-independent, it should be noted that the behavior of JavaScript code may vary across different (versions of) browsers, so slightly different results might be expected on different platforms.

7 RELATED WORK

It has long been known (see, e.g., Saltzer [1966]) that software may exhibit race conditions, i.e., situations where program behavior depends on the nondeterministic ordering of tasks that is not under the control of the programmer. Race conditions are typically considered errors if some but not all orderings result in undesirable program behavior. This problem has been studied in depth for programming languages with shared-memory concurrency (see, e.g., Boyapati and Rinard [2001]; Flanagan and Freund [2000, 2008, 2010]; Hammer et al. [2008]; Naik et al. [2006]; Savage et al. [1997]; Voung et al. [2007]), but races also appear in languages without concurrency that feature asynchronous or event-driven control flow. The remainder of this section focuses on work involving race conditions in event-driven systems, specifically JavaScript web applications.

Detecting Event Races in JavaScript Web Applications. The fact that event race errors occur in JavaScript programs was initially observed by Steen [2009] and Ide et al. [2009]. Zheng et al. [2011] presented the first approach to automatically find such errors, however, it is based on a static analysis that is insufficiently precise to handle real websites. The WebRacer tool by Petrov et al. [2012] instead uses dynamic analysis, based on a JavaScript-specific happens-before relation.

[17] `https://github.com/eth-srl/webkit`

Raychev et al. [2013] observed that the number of event races in a JavaScript web application can be overwhelming, which motivated a notion of race coverage. By focusing on uncovered races, their EventRacer tool dramatically reduces the number of reported races, but it may hide harmful errors. As discussed, EventRacer (as well as its predecessor WebRacer) has several limitations that hinder practical use. For example, EventRacer reports races regardless of whether they may be harmful. It does not account for "long delays" or visibility of HTML elements (Sections 4.1 and 4.2), and it sometimes reports ad-hoc synchronization, which has been inserted into the code to *prevent* race errors, as likely harmful. Furthermore, incomplete modeling of the happens-before relation, which is inevitable due to the rapid development of the browser APIs, leads to more races being reported (Section 4.1).

Another practical problem with EventRacer is that it builds on top of a version of WebKit that is thousands of commits behind the current release, which, as discussed in Section 6 (RQ4), makes it unsuitable for analyzing some modern web pages. InitRacer instead relies on dynamic code instrumentation and is fully platform independent.

EventRacer only detects races in code that has been executed by a given user event sequence, and its auto-exploration only triggers a small fraction of the relevant events, causing it to miss harmful races. For example, it fails to automatically detect the three event race errors in Section 2. In contrast, InitRacer's adverse execution mode explores all registered user event handlers, and its analysis is not limited to races that appear with a specific user event sequence, thus enabling it to find more errors in a single run. Moreover, InitRacer gains precision by identifying long delays in the initialization process and taking HTML element visibility into account.

EventRacer outputs only the trace that contains the races, with no information about how the races may affect the execution, which makes it difficult to diagnose and debug the errors. As discussed in Section 5, InitRacer provides detailed diagnostic information to facilitate debugging.

Diagnosing Event Races. There is an important difference between *races* and the *errors* they may cause. Many races are completely harmless. Despite its attempt to classify races, EventRacer does not have evidence that the two operations involved in a race can in fact be reordered, or that the opposite ordering (if it exists) is harmful. For example, a runtime exception originating from an access-before-definition could be caught by a `catch` block in the program, rendering the race harmless. In general, any tool for detecting race errors must reason about the effects of the individual races, for example, by establishing that only one of the two possible orderings of a race is "good". The fact that many races are harmless has motivated the work discussed below on detecting races that lead to actual errors.

The R^4 tool by Jensen et al. [2015] uses a stateless model checking approach to analyze the entire state space of a nondeterministic web application, relative to a given user event sequence. In addition, it uses a notion of approximate replay to investigate the effect of each race. R^4 explicitly filters away races involving late registration of event handlers and classifies harmfulness of each detected race according to its effect on, e.g., the HTML DOM and uncaught exceptions. In comparison, InitRacer specifically targets the three different types of initialization races described in Section 4.

The technique by Mutlu et al. [2015] is designed for detecting races that affect persistent storage. In their view, any error that may disappear by reloading the web page is considered benign, even though the error may damage functionality or user experience, which makes their technique unsuitable for detecting initialization race errors.

The WAVE tool by Hong et al. [2014] aims to investigate the effect of each race by executing alternative schedulings of the same user event sequence, like R^4 but without using model checking or approximate replay techniques. Previous work has shown that the approach taken by WAVE often results in an overwhelming amount of false positives [Jensen et al. 2015].

The RClassify tool by Zhang and Wang [2017] aims to determine whether a given race is harmful. It uses a replay-based method that forces the execution of a pair of racing events in two different orders and assesses the impact on the program state by comparing the values stored in, for example, the DOM and the JavaScript variables. Similar to InitRacer, RClassify works using instrumentation and is platform independent. However, unlike InitRacer, RClassify requires as input a set of races produced by a separate race detection tool; the experiments reported by Zhang and Wang were based on EventRacer to supply this initial set of races.

In contrast to these event race classification techniques, InitRacer entirely avoids the need for explicitly identifying racing memory accesses and is capable of detecting initialization race errors, with high speed and accuracy, using relatively simple instrumentation techniques.

Automated Repair of Event Races. Experience thus far has been that event races are extremely common, and that many event races have similar characteristics and occur for similar reasons. Beyond detecting races and determining their harmfulness, recent work has focused on automatic repair of web applications, in ARROW [Wang et al. 2016] by reordering script fragments, and in EventRaceCommander [Adamsen et al. 2017] by controlling how event handlers are scheduled for execution. Such techniques are complementary to InitRacer. For example, we believe it is possible to define repair policies for EventRaceCommander that are tailored to the different categories of initialization races errors that are targeted by InitRacer.

8 CONCLUSION

We have presented a simple but effective technique for detecting initialization race errors in JavaScript web applications, and its implementation in a tool called InitRacer. Our technique matches a small number of patterns against the trace of actions performed by a web application, using a three-phase approach to observe actions in different execution modes. Unlike previous techniques, InitRacer is based on dynamic code instrumentation and is therefore platform-independent. Furthermore, InitRacer produces informative error messages to support diagnosing the causes and effects of the races.

In an evaluation on 100 real-world websites, InitRacer reports 1 085 initialization races, while providing informative explanations of their causes and effects. A manual study of 218 of these reports shows that 111 of them lead to uncaught exceptions, although without directly affecting user experience, and at least 47 indicate errors that affect the functionality of the websites.

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), and, in part, by NSF grant CCF-1715153.

REFERENCES

Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic Execution of Android Test Suites in Adverse Conditions. In *Proc. 24th International Symposium on Software Testing and Analysis (ISSTA)*. 83–93.

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)*.

Esben Andreasen and Anders Møller. 2014. Determinacy in Static Analysis for jQuery. In *Proc. International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. 17–31.

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.

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.

Cormac Flanagan and Stephen N. Freund. 2008. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. *Sci. Comput. Program.* 71, 2 (2008), 89–109.

Cormac Flanagan and Stephen N. Freund. 2010. FastTrack: Efficient and Precise Dynamic Race Detection. *Commun. ACM* 53, 11 (2010), 93–101.

Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *Proc. 24th International Symposium on Software Testing and Analysis (ISSTA).* 94–105.

Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. 2008. Dynamic Detection of Atomic-Set-Serializability Violations. In *Proc. 30th International Conference on Software Engineering (ICSE).* 231–240.

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.

Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. 2014. Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor. In *Proc. 9th Eurosys Conference.* 18:1–18:15.

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

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.

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.

Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Proc. ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI).* 308–319.

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.

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.

Jerome Howard Saltzer. 1966. *Traffic Control in a Multiplexed Computer System.* Ph.D. Dissertation. Massachusetts Institute of Technology. MAC-TR-30.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (1997), 391–411.

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

Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: Static Race Detection on Millions of Lines of Code. In *Proc. 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE).* 205–214.

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.

Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200.

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).*

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).*