Repairing Event Race Errors by Controlling Nondeterminism

Christoffer Quist Adamsen*
Anders Møller
Aarhus University
Aarhus, Denmark
{cqa,amoeller}@cs.au.dk

Rezwana Karim
Manu Sridharan†
Samsung Research America
Mountain View, CA, USA
rezwana.k@samsung.com
manu@sridharan.net

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

Koushik Sen
EECS Department
UC Berkeley, CA, USA
ksen@cs.berkeley.edu

***Abstract*—Modern web applications are written in an event-driven style, in which event handlers execute asynchronously in response to user or system events. The nondeterminism arising from this programming style can lead to pernicious errors. Recent work focuses on detecting event races and classifying them as harmful or harmless. However, since modifying the source code to prevent harmful races can be a difficult and error-prone task, it may be preferable to steer away from the bad executions.**

In this paper, we present a technique for automated repair of event race errors in JavaScript web applications. Our approach relies on an event controller that restricts event handler scheduling in the browser according to a specified repair policy, by intercepting and carefully postponing or discarding selected events. We have implemented the technique in a tool called *EventRaceCommander*, which relies entirely on source code instrumentation, and evaluated it by repairing more than 100 event race errors that occur in the web applications from the largest 20 of the Fortune 500 companies. Our results show that application-independent repair policies usually suffice to repair event race errors without excessive negative impact on performance or user experience, though application-specific repair policies that target specific event races are sometimes desirable.

***Keywords*-JavaScript; event-driven programming; automated repair**

I. Introduction

Modern application development has largely moved to platforms requiring event-driven programming, using web browsers and mobile platforms. The event-driven model is well-suited to the needs of today's interactive programs, which must perform high-latency network requests to send and receive requested data while remaining responsive to user input. However, as studied in recent work [8, 11, 21, 22, 25, 34], this programming style can cause pernicious nondeterminism errors, which can lead to crashes, lost user data, and malfunctioning user interfaces.

Recent work has attacked this nondeterminism problem through tools for detecting *event races*, where application behavior may differ depending on the order in which event handlers are executed. For web applications, event race detectors are capable of finding errors in real-world, deployed web applications [25]. Further, tools such as R^4 [11] can filter away warnings about races that do not affect the external behavior of web applications.

Despite these advances, the output of state-of-the-art event race detectors is often still not practical. Diagnosing the root cause of an event race in a real-world web application can require a significant effort—it often requires deciphering complex event sequences, and it can be difficult to classify how harmful a reported race is, especially for non-expert users of the tools. In addition, preventing such races may require introducing complex synchronization into the code, an arduous task since the web platform provides few mechanisms for synchronizing across event handlers. Manually devising such a fix is often not worth the effort, particularly for minor errors, when considering that fixes sometimes have unforeseen consequences [31]. Better techniques are needed to reduce the cost of fixing event race errors.

In this work, we explore *automated repair* of event race errors in web applications. Automated repair holds great promise for addressing the aforementioned drawbacks of event race detectors. If event race errors can be automatically repaired, without requiring developers to deeply understand root causes, the errors may be avoided more often. There is a wide body of work on repairing races in multi-threaded programs [6, 12–18, 23, 24, 27, 29, 30, 32, 33], but relatively little work on repair for event races. Wang et al. [28] have proposed a repair technique for event races in web applications, but it has significant limitations in the types of races it can handle (see Section VIII).

Our approach builds on an *event controller* that restricts event handler scheduling in the browser according to a specified *repair policy*, by intercepting and carefully postponing or discarding selected events. Restricting schedules dynamically to avoid bad orderings is a well-known approach to automated repair of races in the context of shared-memory concurrency races, but to our knowledge it has not previously been applied to event-driven applications. An important property of our approach is that the event controller is built entirely by instrumenting the web application code. Most event race detection tools for JavaScript work using modified browsers, which is reasonable for detecting races, but not for automated repair,

*The work of this author was carried out during an internship at Samsung Research America.
†The author's current affiliation is Uber.

as the code must run on end-user browsers. In spite of relying solely on light-weight instrumentation, our approach is general enough to repair common types of event races, although some event race errors cannot be repaired by merely restricting the nondeterminism (see Section V-D).

Given this event controller, the question remains of what policies are required to repair races in practice. A policy specifies which events to postpone or discard, so choosing an appropriate policy requires knowledge about which event orders are good and which are bad. We observe that many races in web applications can be prevented with a small collection of *application-independent* policies. For example, application developers often expect Ajax response handlers to execute in a first-in first-out (FIFO) order, and that the page completes loading before the user interacts with it: many errors occur when these assumptions are violated. Our application-independent policies enforce these assumptions, yielding a simple method for avoiding many races.

Application-independent policies are easy to apply, but may impact performance or user experience negatively. For example, delaying all user events until after the page has loaded may make the page appear sluggish, and in fact many user interactions during page load may be perfectly safe (i.e., they cannot lead to harmful races). We show that these problems can be alleviated using application-specific policies, which can be designed, for example, by specializing an application-independent policy to a particular web application.

In summary, the contributions of this paper are as follows.

- We demonstrate that most errors involving event races in JavaScript web applications can be repaired automatically, using light-weight instrumentation to steer the nondeterminism according to a specified repair policy.
- We propose the use of application-independent policies, which can be specialized as needed to avoid excessive delay in event processing, or to target specific event races reported by existing race detection tools.
- We evaluate our approach based on an implementation called *EventRaceCommander*, by repairing 117 event race errors in the websites of the 20 largest companies from the Fortune 500 list. Our results show that 94 of the errors can be repaired using application-independent policies, mostly without excessive negative impact, and that application-specific policies can alleviate the undesirable effects when this is not the case.

II. Motivating Example

Figure 1 shows a small web application for browsing through galleries of images, consisting of three files. File `index.html` defines a top-level page with two buttons, labeled "Gallery 1" and "Gallery 2." Clicking each button causes function `loadThumbs` (lines 15–26) to be invoked with the gallery name "`g1`" or "`g2`," depending on the gallery being selected. Executing `loadThumbs` will send an Ajax request to the server (lines 17–25). When the server responds, the `readystatechange` callback function (lines 18–23) is invoked. This callback parses the response to retrieve an array of thumbnail images and stores them in variable `thumbs` (line 20), and then invokes `showThumbs` with the same gallery name as before. Function `showThumbs` (lines 27–40) iterates through `thumbs` and creates a '`Delete`' button for each image that, when clicked, will invoke `deleteImg` with the gallery name and index of the image. Function `deleteImg` (lines 41–46) creates another Ajax request, requesting the selected image to be deleted from the server (lines 43–44).

index.html

```
 1 <html>
 2   ...
 3   <div id="container" ...>
 4   ...
 5   <button id="g1">Gallery 1</button>
 6   <button id="g2">Gallery 2</button>
 7   <script src="init.js"></script>
 8   <script src="script.js"></script>
 9 </html>
```

init.js

```
10 document.getElementById('g1').addEventListener(
11   'click', function () { loadThumbs('g1'); }, false);
12 document.getElementById('g2').addEventListener(
13   'click', function () { loadThumbs('g2'); }, false);
```

script.js

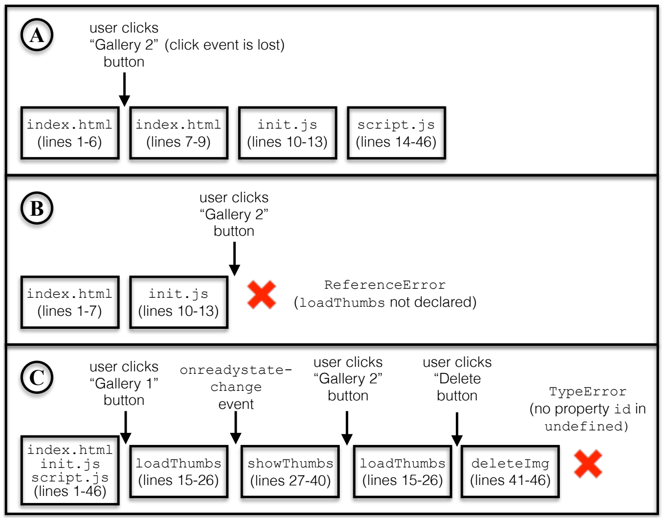
```
14 var thumbs;
15 function loadThumbs(name) {
16   thumbs = [];
17   var xhr = new XMLHttpRequest();
18   xhr.onreadystatechange = function () {
19     if (xhr.readyState === XMLHttpRequest.DONE) {
20       thumbs = JSON.parse(xhr.responseText);
21       showThumbs(name);
22     }
23   };
24   xhr.open('GET', 'gallery?name=' + name, true);
25   xhr.send(null);
26 }
27 function showThumbs(name) {
28   container.innerHTML = '';
29   for (var pos = 0; pos < thumbs.length; ++pos) {
30     ... // display thumbnail image
31     var b = document.createElement('button');
32     b.textContent = 'Delete';
33     (function (pos) {
34       b.addEventListener('click', function (e) {
35         deleteImg(name, pos);
36       }, false);
37     })(pos);
38     container.appendChild(b);
39   }
40 }
41 function deleteImg(name, pos) {
42   ...
43   xhr.open('POST', 'gallery?action=delete&name='
44       + name + '&img=' + thumbs[pos].id, true);
45   ...
46 }
```

Fig. 1. Motivating example (inspired by Zheng et al. [34]).

A. Event Races

The example application exhibits three event races that may cause runtime exceptions or other unexpected behavior, depending on the order in which event handlers execute. The corresponding undesirable schedules are illustrated in Figure 2 and discussed in detail below.

Ⓐ If the user clicks the "Gallery" buttons before `init.js` has executed, then the user event is lost, since the `click` event handlers are not yet registered.



A
user clicks "Gallery 2" button (click event is lost)
index.html (lines 1-6)
index.html (lines 7-9)
init.js (lines 10-13)
script.js (lines 14-46)
B
user clicks "Gallery 2" button
index.html (lines 1-7)
init.js (lines 10-13)
ReferenceError (loadThumbs not declared)
C
user clicks "Gallery 1" button
onreadystate-change event
user clicks "Gallery 2" button
user clicks "Delete button
TypeError (no property id in undefined)
index.html init.js script.js (lines 1-46)
loadThumbs (lines 15-26)
showThumbs (lines 27-40)
loadThumbs (lines 15-26)
deleteImg (lines 41-46)

Fig. 2. Illustration of event races in the program of Figure 1.

Ⓑ If the user clicks the "Gallery" buttons after executing `init.js`, but before `script.js` has executed, then an event handler is associated with the `click` event, but function `loadThumbs` is still undeclared. Hence, executing either of the `click` event handlers on lines 11 and 13 triggers an uncaught `ReferenceError`.

Ⓒ Assume the user clicks the "Gallery 1" button after the entire page has been parsed. This causes `loadThumbs` (lines 15–26) to execute with argument "`g1`," generating an Ajax request. When the server responds, the event handler on lines 18–23 executes, causing `showThumbs` to execute (lines 27–40). If the user then clicks the "Gallery 2" button, `loadThumbs` runs again (now with argument "`g2`") assigning an empty array to `thumbs` before making a second Ajax request. Now, say the user clicks the "Delete" button for an image that is still on the screen, before the response to the second Ajax request is received. Then, the `click` handler on lines 34–36 invokes `deleteImg` (lines 41–46), causing the expression `thumbs[pos].id` to be evaluated (line 44). But `thumbs` is still empty! So, `thumbs[pos]` evaluates to `undefined`, and accessing the `id` property of `undefined` yields an uncaught `TypeError`.

We will refer to scenarios where user events interfere with initializations performed during page loading (e.g., scenarios Ⓐ and Ⓑ) as *initialization races*. Races such as the one in scenario Ⓒ will be referred to as *post-initialization races*.

B. Repairing Event Race Errors

The types of problems discussed above commonly occur when a schedule differs from developers' expectations. For example, developers typically test their code in environments where the parsing and loading of a page is fast and where user actions do not occur until page loading is complete. Scenarios like Ⓐ and Ⓑ violate this assumption, causing various sorts of errors to arise when user events arrive at inopportune moments. Similarly, developers commonly assume the network to be fast, so that responses to Ajax requests are received before the user performs additional actions. Scenario Ⓒ, where the user clicks on "Delete" before the response for the click on "Gallery 2" is received, violates this assumption, resulting in a runtime error.

Our approach for preventing undesirable schedules relies on code instrumentation, and takes as input a repair policy that specifies constraints on the scheduling of event handlers. In particular, the web application is instrumented so that all events are intercepted and monitored by a runtime controller. At runtime, when an event arrives that is not in accordance with the repair policy, it is either discarded or postponed until the execution of the associated event handlers agrees with the policy. For example, scenarios Ⓐ and Ⓑ can be prevented in our approach by an application-independent policy that postpones user events until all statically declared scripts are loaded, by intercepting the events and regenerating them later. (In cases where this policy blocks harmless user events, one can easily create a policy that only postpones clicks on the "Gallery" buttons.) Likewise, scenario Ⓒ can be prevented by an application-independent policy that discards user events after an Ajax request until the response arrives. In such cases, *EventRaceCommander* shows a "spinner" on the screen to inform the end-user that user events are temporarily blocked.

While the three scenarios discussed here can be repaired using application-independent policies, application-specific policies may be preferable, as we shall see in Section VII.

III. Background on Event Races

This section defines event races and related concepts using a simplified version of the formalism of Raychev et al. [25].

We instrument an event-based program to generate a sequence of *operations*, called a *trace*, for a given execution. An operation can be of the following kinds (assuming each event is given a unique identifier u):

- $\mathit{read}(u, x)$ and $\mathit{write}(u, x)$ denote that an event handler of u reads and writes, respectively, a shared variable x.
- $\mathit{fork}(u, v)$ denotes that an event handler of u creates a new event v that will be dispatched later.
- $\mathit{begin}(u)$ and $\mathit{end}(u)$ denote the beginning and ending, respectively, of the execution of u's event handlers.

We denote the set of all operations by Op, and the event to which an operation σ belongs by $\mathit{evt}(\sigma)$. The execution of a program generates a finite trace $\tau = \sigma_0 \cdots \sigma_n \in \mathit{Op}^*$. In event-based programs, all event handlers of an event execute atomically without interleaving with the execution of any handler of another event. Therefore, if an event u gets dispatched, then all the operations from the event handlers of u appear as a contiguous subsequence in the trace, where the first and last operations of the subsequence are $\mathit{begin}(u)$ and $\mathit{end}(u)$, respectively. If the trace contains an operation $\mathit{fork}(u, v)$, then $\mathit{begin}(u)$ appears before $\mathit{begin}(v)$, i.e., an event must be created before it gets dispatched.

A trace τ defines a linear relation $<$, where $\sigma < \sigma'$ if the operation σ appears before σ' in the trace τ. As in traditional concurrent programs, we can define a happens-before relation $\preceq$ as the minimal partial order (i.e., a reflexive, anti-symmetric, transitive relation) over the events of a trace such that $u \preceq v$ if $\mathit{fork}(u, v) \in \tau$ or if u and v are user events where $\mathit{begin}(u) <$

$$\begin{aligned}&\mathit{scheduler}(\sigma,\tau,\mathcal{P}) := (\tau\cdot \mathit{extend}(\sigma,\tau,\mathcal{P}), \mathit{update}(\sigma,\tau,\mathcal{P}))\\ &\mathit{update}(\sigma,\tau,\mathcal{P}) := \mathcal{P}\setminus\{(q,s,1,a,r)\in\mathcal{P}_A(\sigma,\tau,S_\tau)\}\\ &\quad\bigcup\{r_i\in r(\sigma)\mid (q,s,t,a,r)\in\mathcal{P}_A(\sigma,\tau,S_\tau)\}\end{aligned}$$

$$\mathit{extend}(\sigma,\tau,\mathcal{P}) := \begin{cases}\mathit{discard}(u) & \text{if } \sigma\equiv \mathit{begin}(u)\wedge \mathit{action}(u,\tau,\mathcal{P})\equiv \text{DISCARD}\\ \mathit{postpone}(u) & \text{if } \sigma\equiv \mathit{begin}(u)\wedge \mathit{action}(u,\tau,\mathcal{P})\equiv \text{POSTPONE}\\ \sigma & \text{otherwise}\end{cases}$$

$$\mathit{action}(u,\tau,\mathcal{P}) := \max\{a\mid (q,s,t,a,r)\in\mathcal{P}_A(\mathit{begin}(u),\tau,S_\tau)\}, \text{where } \text{DISPATCH} < \text{POSTPONE} < \text{DISCARD}$$

Fig. 3. The effect of a repair policy on the execution.

$\mathit{begin}(v)$. Two events u and v are unordered, denoted by $u \parallel v$, if they are not related by the happens-before relation. We are now in a position to define the notion of an event race.

An *event race* (σ, σ') is a pair of operations from a trace τ where $\sigma < \sigma'$, $\mathit{evt}(\sigma) \parallel \mathit{evt}(\sigma')$, both σ and σ' access (i.e., read or write) the same shared variable, and at least one of σ and σ' is a write.

Recent work has focused on classifying event races as either harmful or harmless [11, 20, 21, 25]. In general, such classification is a subjective matter. In many cases, lost user events or uncaught exceptions do not significantly affect the user experience and do not require remediation, though for some web sites such errors are intolerable. Our approach side-steps this ambiguity by relying on the user of *EventRaceCommander* to distinguish desirable schedules from undesirable ones, and applying a repair policy that prevents undesirable schedules from taking place. In other words, our approach does not rely on a particular definition of harmfulness, nor is it limited to races that are considered harmful.

IV. A Framework for Specifying Repair Policies

This section presents a framework for constraining the order in which event handlers are executed using a specified repair policy. A *repair policy* $\mathcal{P}$ consists of a set of rules, which upon their activation (determined by the current trace τ and program state S_τ of the web application) may discard or postpone events occurring in the execution. To this end, we add the following types of operations.

- $\mathit{discard}(u)$ denotes the discarding of event u (i.e., the event handlers for event u will not be invoked, and no $\mathit{begin}(u)$ operation will ever appear in the trace).
- $\mathit{postpone}(u)$ denotes the postponement of event u (i.e., u will be re-emitted later, and at least one of the operations $\mathit{begin}(u)$, $\mathit{discard}(u)$, $\mathit{postpone}(u)$ will appear in the trace once u is re-emitted).

A key contribution of our work is that we—based on a study of many event races in real-world web applications—observe that harmful races mostly arise for similar reasons, and can be repaired using *application-independent* policies.

A *rule* is a quintuple of the form $(q, s, t, a, r) \in \mathcal{Q}$ where:

- q is an *operation predicate* over the operations in *Op* that specifies a necessary condition for the rule to be activated upon the current operation of a scheduler.
- s is an *expiration status*, which is a predicate over the pairs of traces and program states that determines if the rule is enabled or expired.
- $t \in \{1, \infty\}$ is a *scope*. If $t = 1$, then the rule expires the first time it is activated, otherwise it remains enabled until the expiration status s becomes true.
- $a \in \{\text{DISPATCH}, \text{DISCARD}, \text{POSTPONE}\}$ is an *action* in response to the event that the rule was activated by.
- $r : \mathit{Op} \to 2^{\mathcal{Q}}$ is an *extension function* that maps an operation to a policy, which is used for dynamically adding new rules to the existing policy.

For the sake of presentation, we will use $(q, s, t, a) \overset{\sigma}{\Rightarrow} r_0, \ldots, r_n$ to denote the policy $\{(q, s, t, a, \lambda\sigma. \bigcup_{0\leq i\leq n} r_i)\}$, and (q, s, t, a) to denote the policy $\{(q, s, t, a, \varnothing)\}$.

A rule $(q, s, t, a, r) \in \mathcal{P}$ is *activated* by an operation σ in state (τ, S_τ) if $q(\sigma)$ and $\neg s(\tau, S_\tau)$ hold, i.e., the rule matches the operation and is not expired. We denote by $\mathcal{P}_A(\sigma, \tau, S_\tau)$ the set of rules in $\mathcal{P}$ that are activated by σ in (τ, S_τ). The definition of activated rules enables us to describe the effect of a repair policy on the execution by means of a function, $\mathit{scheduler}(\sigma, \tau, \mathcal{P})$ (Figure 3), that maps an operation, a trace, and a repair policy to an extended trace and updated policy. The auxiliary function $\mathit{extend}(\sigma, \tau, \mathcal{P})$ (Figure 3) determines whether events should be discarded or postponed, by computing the action for σ as a maximum over the activated actions (multiple rules may be activated simultaneously). The ordering among actions is defined in Figure 3. If σ is not a *begin* operation, then τ is simply extended with σ. Hence, a policy cannot discard or postpone an event based on specific operations within an event handler. Rules can, however, match on specific operations and use them to modify the policy via the extension function.

In addition to extending the trace τ, the current repair policy $\mathcal{P}$ is replaced by $\mathcal{P}' = \mathit{update}(\sigma, \tau, \mathcal{P})$ (Figure 3), which differs from $\mathcal{P}$ as follows.

1) All rules with scope 1 that were activated by σ in (τ, S_τ) are removed from $\mathcal{P}$.
2) $\mathcal{P}$ is extended with the rules in $r(\sigma)$ for every activated rule (q, s, t, a, r).

We emphasize that postponing one event may require other events to be postponed as well, due to the happens-before relation of the original web application. For example, the `load` event of a script always follows the execution of the same script. Our framework automatically enforces such constraints, and additionally preserves the order of user events.

V. Repair Policies

We identify five classes of event races that cause many problems in practice. Section V-A defines *application-independent* policies in terms of the framework presented in Section IV. Then, Section V-B shows how such general policies can be specialized to particular web applications, for improved performance and user experience.

$$
\begin{aligned}
q_{\mathit{user}}(\sigma) &:= \; \sigma = \mathit{begin}(u) \wedge \mathit{type}(u) \in \{\texttt{keydown}, \texttt{mousedown}, \dots\} \\
q_{\mathit{callback}}(\sigma) &:= \; \sigma = \mathit{begin}(u) \wedge (\mathit{type}(u) = \texttt{timer} \vee \\
&\qquad (\mathit{type}(u) = \texttt{load} \wedge \mathit{tagName}(\mathit{target}(u)) \in \{\texttt{iframe}, \texttt{img}\})) \\
q_{\mathit{fork}}(\sigma) &:= \; \sigma = \mathit{fork}(\cdot, v) \wedge (\mathit{type}(v) \in \{\texttt{script-exec}, \texttt{timer}\} \vee \\
&\qquad (\mathit{type}(v) = \texttt{readystatechange} \wedge \mathit{readyState}(\mathit{target}(v)) = 4)) \\
q_{\mathit{begin}}(u, \sigma) &:= \; \sigma = \mathit{begin}(u)
\end{aligned}
$$

(a)

$$
\begin{aligned}
\textsc{Arrived}(u, \tau, S_\tau) &:= \; \mathit{begin}(u) \in \tau \\
\textsc{Parsed}(\tau, S_\tau) &:= \; \text{DOM}\textsc{ContentLoaded} \in \textstyle\bigcup_{\mathit{begin}(u) \in \tau} \mathit{type}(u)
\end{aligned}
$$

(b)

$$
\begin{aligned}
\textsc{WaitFor}(u) &:= \; (q_{\mathit{user}}, \textsc{Arrived}(u), \infty, \textsc{Discard}) \\
\textsc{WaitRec}(u) &:= \; (q_{\mathit{fork}}, \textsc{Arrived}(u), \infty, \textsc{Dispatch}) \\
&\qquad \xRightarrow{\mathit{fork}(v,w)} \textsc{WaitFor}(w), \textsc{WaitRec}(w) \\
\textsc{Order}(u, v) &:= \; (q_{\mathit{begin}}(v), \textsc{Arrived}(u), \infty, \textsc{Postpone}) \\
\textsc{OrderNext}(u) &:= \; (q_{\mathit{fork}}, \textsc{Arrived}(u), 1, \textsc{Dispatch}) \\
&\qquad \xRightarrow{\mathit{fork}(v,w)} \textsc{Order}(u, w)
\end{aligned}
$$

(c)

$$
\begin{aligned}
\mathcal{P}_{\mathit{init,user}} &:= \; (q_{\mathit{user}}, \textsc{Parsed}, \infty, \textsc{Postpone}) \\
\mathcal{P}_{\mathit{init,system}} &:= \; (q_{\mathit{callback}}, \textsc{Parsed}, \infty, \textsc{Postpone}) \\
\mathcal{P}_{\mathit{async,user}} &:= \; (q_{\mathit{fork}}, \top, \infty, \textsc{Dispatch}) \xRightarrow{\mathit{fork}(u,v)} \textsc{WaitFor}(v) \\
\mathcal{P}_{\mathit{async,fifo}} &:= \; (q_{\mathit{fork}}, \top, \infty, \textsc{Dispatch}) \xRightarrow{\mathit{fork}(u,v)} \textsc{OrderNext}(v) \\
\mathcal{P}^{+}_{\mathit{init,user}} &:= \; \mathcal{P}_{\mathit{init,user}} \bigcup \Big((q_{\mathit{fork}}, \textsc{Parsed}, \infty, \textsc{Dispatch}) \\
&\qquad \xRightarrow{\mathit{fork}(u,v)} \textsc{WaitFor}(v), \textsc{WaitRec}(v) \Big)
\end{aligned}
$$

(d)

Fig. 4. Repair policies. **(a)** operation predicates, **(b)** expiration status utilities, **(c)** utility functions, **(d)** application-independent repair policies.

A. Application-Independent Repair Policies

User events before **`DOMContentLoaded`*****:*** Scenarios Ⓐ and Ⓑ from Section II-A illustrate initialization races that lead to undesirable behavior when a user interacts with a web page before it has been fully parsed. The errors induced by these races can be repaired by enforcing the policy $\mathcal{P}_{\mathit{init,user}}$ from Figure 4(d), where q_{user} is an operation predicate that matches any user event. Due to the definition of the policy's expiration status, $\textsc{Parsed}$ (Figure 4(b)), this policy postpones any user event until the event handlers of `DOMContentLoaded` have been executed. It is easy to see how this policy prevents the errors in scenarios Ⓐ and Ⓑ from Section II-A: By preventing `click` events on the "Gallery" buttons until the page has been parsed, the `click` event handlers will be registered in time, and the `loadThumbs` function will be defined before it is invoked, thereby preventing the `ReferenceError`.

In this policy, $\textsc{Discard}$ could be used instead of $\textsc{Postpone}$. The $\textsc{Discard}$ action is intended for user events only, since users can always simply repeat their inputs when the policy allows it, which is not possible for system events.

System events before **`DOMContentLoaded`*****:*** Harmful initialization races also arise when system events fire unexpectedly early. In the following example, which is based on code from `exxon.com`, the `load` event listener attached by the script will never run if the `iframe` loads prior to the execution of the script.

```
47 <iframe src="..." id="iframe"></iframe>
48 ...
49 <script>
50 $('#iframe').load(function (e) { /* adjust iframe height */ });
51 </script>
```

Such errors can be repaired using the policy $\mathcal{P}_{\mathit{init,system}}$ from Figure 4(d), which postpones system events, such as the `load` event of the `iframe` in line 47, until the page has been parsed. $\mathcal{P}_{\mathit{init,system}}$ matches any `iframe` or `img` `load` event, and any timer event, with the operation predicate q_{callback}.

User events while async event is pending: Scenario Ⓒ in Section II-A represents a situation where the application logic implicitly assumes that asynchronously forked events are handled atomically, without being interrupted by user events. Such post-initialization race errors can be prevented using policy $\mathcal{P}_{\mathit{async,user}}$ of Figure 4(d). Informally, this policy adds the rule $\textsc{WaitFor}(v)$ (Figure 4(c)) to the policy whenever an operation forks an asynchronous event v (e.g., Ajax request, asynchronous script request, `setTimeout`). This rule discards user events until v is observed in the trace.

Ajax FIFO: Sometimes programmers implicitly assume that the responses to multiple Ajax requests arrive in the same order as the requests were made. Consider the following example, which captures the essence of a race from `gazzetta.it` [21]:

```
ajax('POST', url1, function (a) { document.cookie = f(a); });
ajax('POST', url2, function (b) { document.cookie = g(b); });
```

The two callback functions are executed in response to the first and second Ajax request, respectively. Both functions assign some data from the server's response to the same `document.cookie` key. Therefore, the value of this key depends on the order in which Ajax responses arrive.

To prevent such races, we use policy $\mathcal{P}_{\mathit{async,fifo}}$ of Figure 4(d) to postpone Ajax response events that would break FIFO order: Upon each Ajax request operation $\mathit{fork}(\cdot, v)$, the policy starts listening for the next Ajax request operation $\mathit{fork}(\cdot, w)$ by adding the rule $\textsc{OrderNext}(v)$. The use of scope 1 in $\textsc{OrderNext}$ ensures that the rule will not be activated upon any Ajax request operations following $\mathit{fork}(\cdot, w)$. If $\mathit{begin}(v)$ appears in the trace before $\mathit{fork}(\cdot, w)$, then FIFO is already maintained and $\textsc{OrderNext}(v)$ expires due to its expiration status, $\textsc{Arrived}(v)$. Otherwise, FIFO is enforced by $\textsc{Order}(v, w)$ (added from $\textsc{OrderNext}(v)$), which postpones $\mathit{begin}(w)$ until $\mathit{begin}(v)$ appears in the trace. Furthermore, $\textsc{OrderNext}(w)$ is added (by the rule in $\mathcal{P}_{\mathit{async,fifo}}$) to order $\mathit{begin}(w)$ with the response of the Ajax request operation that follows $\mathit{fork}(\cdot, w)$ (if any).

User events before async initialization: Sometimes initialization actions are being performed by asynchronously executed code. Consider the following snippet, which was extracted from `flysas.com`.

```
<input id="from-airport" /><input id="to-airport" />
<script>
```

```
56 var lastFrom = ..., lastTo = ...; // inspect cookie
57 $.get('/service?code=' + lastFrom, function (from) {
58   $.get('/service?code=' + lastTo, function (to) {
59     $('#from-airport').val(from.name);
60     $('#to-airport').val(to.name);
61   });
62 });
63 </script>
```

During loading, the user's input may be overwritten, since the fields in lines 59–60 are not initialized until the responses of the two Ajax requests in lines 57–58 have been processed. This may happen after the DOMContentLoaded event, and therefore the policy $\mathcal{P}_{init,user}$ does not suffice to repair the race. To accommodate for this, we define an extension of this policy, $\mathcal{P}^{+}_{init,user}$, that additionally discards user events until asynchronous initialization has been performed.

Intuitively, $\mathcal{P}^{+}_{init,user}$ continuously adds $\textsc{WaitFor}(v)$ (which discards user events until $begin(v)$ appears in the trace) for every operation $fork(\cdot, v)$ that matches q_{fork}, as long an event that has been forked by some other operation matching q_{fork} is pending. For example, if $fork(\cdot, v)$ and $fork(\cdot, w)$ denote the Ajax requests in lines 57 and 58, respectively, then $\textsc{WaitFor}(v)$ is added upon $fork(\cdot, v)$, which discards user events until the callback in lines 57–62 has executed. In addition, $\textsc{WaitRec}(v)$ is added, which itself adds $\textsc{WaitFor}(w)$ upon $fork(\cdot, w)$. The $\textsc{WaitFor}(w)$ rule discards user events until after the callback in lines 58–61.

The WaitRec rule recursively adds new rules to approximate when asynchronous initialization is over. This may lead to user events being discarded indefinitely (e.g., in the presence of image sliders that keep changing automatically). Thus, this policy should only be used for pages that always "terminate" (i.e., where the event queue eventually becomes empty if no more user events are made), or q_{fork} should be defined such that it excludes operations that are not part of initialization (e.g., by ignoring timer operations).

B. Application-Specific Policies

The application-independent policies can be applied without a deep understanding of the races, and suffice for preventing the majority of the race errors (see Section VII). However, sometimes the policies negatively affect web page responsiveness (e.g., the user experience of a web page can be degraded when too many user events are interrupted). This motivates *application-specific* repair policies that reduce disruption. It is straightforward to refine an application-independent policy to specific user events. The manual effort required to design an "optimal" application-specific policy naturally requires understanding the cause of the race.

Specializing an application-independent policy to a concrete web application is straightforward. As an example, recall that the race errors exposed by scenarios Ⓐ and Ⓑ can be prevented by enforcing the policy $\mathcal{P}_{init,user}$. However, this may unnecessarily affect clicks to buttons other than "Gallery 1" and "Gallery 2" during page loading. This problem can be alleviated by refining the operation predicate q_{user} in $\mathcal{P}_{init,user}$ to only match `click` events on those two buttons.

The interruption of the user is still not minimal, though, since the function `loadThumbs` (Figure 1, lines 15–26) is declared strictly before the `DOMContentLoaded` event gets dispatched. This can be remedied by, for example, exchanging the policy's expiration status from Parsed to one that, unlike all of the application-independent policies, relies on the actual program state to return true when `loadThumbs` is declared in the global scope of S_τ. With this modification, it not only becomes clear that the policy covers the event races in question; it also minimizes the interruption of the user.

C. Effectiveness of Repair Policies

To understand if a repair policy $\mathcal{P}$ prevents the bad order of an event race, recall that state-of-the-art dynamic race detectors, such as EventRacer [25], report event races as two operations σ and σ' in a trace τ where $evt(\sigma) \parallel evt(\sigma')$.

Simply checking that the race disappears when running a race detector on the instrumented program that enforces $\mathcal{P}$ is too naive, since state-of-the-art race detectors are currently unable to reason about ad-hoc synchronization and will report (σ, σ') as a false positive. On the other hand, checking that the race becomes *covered* [25] gives false confidence.[1] Indeed, most races become covered in the instrumented program, since the execution of event handlers is controlled by ad-hoc synchronization in the instrumented program.

To see how a repair policy $\mathcal{P}$ prevents the bad order of (σ, σ'), consider that the instrumentation restricts the non-determinism in the original program, by enforcing an order among certain events in the execution. Assuming that the trace τ obtained by the race detector is valid according to $\mathcal{P}$, it is possible to model the effect of $\mathcal{P}$ by defining an augmented happens-before relation $\preceq_{\mathcal{P}}$ as the minimal partial order such that $u \preceq_{\mathcal{P}} v$ if either $u \preceq v$ or $\mathcal{P}$ would enforce u to execute before v. Using this relation, it is possible to tell if $\mathcal{P}$ would prevent the race (σ, σ') by checking if $\sigma \preceq_{\mathcal{P}} \sigma'$ or $\sigma' \preceq_{\mathcal{P}} \sigma$, giving developers a way to automatically repair races that has been reported from dynamic race detectors (for a fixed catalogue of policies). The relation $\preceq_{\mathcal{P}}$ can be built for multiple application-independent policies by extending EventRacer. It remains open for future work to construct the relation for arbitrary policies.

D. Discussion of Limitations and Liveness

Although it is not a problem for the repair policies we have presented so far, there is a risk for postponing events indefinitely, thereby breaking liveness, when enforcing policies. Generally, we want to prevent some bad ordering $v \cdots u$ by discarding or postponing v until u has been dispatched. To avoid breaking liveness, it must be known by the time v is about to fire that u will inevitably occur later in the execution.

Intuitively, repair policies can only make decisions based on past events and not on future events. Let F be a set of events that are known to happen in the future. Initially, F contains events that always happen during page loading, e.g.,

[1]Intuitively, a race (σ, σ') is *covered* by another race (δ, δ') if (σ, σ') is no longer a race when (δ, δ') is being treated as synchronization.

`DOMContentLoaded`. During execution, as soon as some event is known to happen in the future (e.g., a timeout is registered or an Ajax request is sent), it is added to F. Perhaps surprisingly, F may also contain some user events, since a single user event is typically composed of a sequence of low-level events (e.g., a `keyup` event always follows a `keydown` event). We now define a necessary condition for being able to enforce an order $u \cdots v$: If v comes before u, and $u \notin F$, then there is no way to steer away from the bad execution without potentially breaking liveness, since it is unknown if u will ever arrive (safety can be preserved, though, by postponing v until u, or indefinitely if u never arrives). Otherwise, if we can define (*i*) an operation predicate that identifies *begin*(v), and (*ii*) a state predicate that becomes false at some point after u has been dispatched, then the desired ordering can be enforced.

We call a repair policy *enforceable* for a program if it does not break liveness *in any execution*. Conversely, we call a race *repairable* if there exists an enforceable policy that prevents the bad order of that race. The application-independent policies $\mathcal{P}_{\mathit{init,user}}$, $\mathcal{P}_{\mathit{init,system}}$, $\mathcal{P}_{\mathit{async,fifo}}$, and $\mathcal{P}_{\mathit{async,user}}$ are enforceable for all programs, and $\mathcal{P}^{+}_{\mathit{init,user}}$ is enforceable for all programs that "terminate" (see Section V-A).

There are situations where it is not possible to prevent an ordering $v \cdots u$ by only discarding or postponing events. Consider the following example:

```
64 <script>setTimeout(function () { d = document; }, 0);</script>
65 <script>console.log(d.querySelectorAll('*').length);</script>
```

Here, the callback in line 64 is supposed to execute prior to the script in line 65. If the latter executes first, then the only possible repair is to postpone its execution. However, this will change program behavior, since line 65 counts the number of elements currently in the DOM. We have not seen any such examples in practice, and hypothesize that this situation is rare.

In other cases, although it is technically possible to repair an event race error, the result would have such a negative impact on user experience that we do not consider it. These races involve event handlers that are triggered when the user merely moves the cursor (e.g., `mouseenter`). Using a repair policy, the user can be provided with feedback that the page is not ready. However, for this kind of "indirect" user event (as opposed to mouse clicks and key events), the event handler registration should rather be performed earlier by changing the code.

VI. Implementation

Our implementation, named *EventRaceCommander*, instruments HTML and JavaScript source files of a given web application on-the-fly using *mitmproxy* [3]. The instrumentation intercepts relevant operations and interacts with the event controller, which is loaded before any application code, such that instrumentation and application code do not race.

The implementation of *EventRaceCommander* is available at `https://github.com/cs-au-dk/EventRaceCommander`.

A. Controlling the execution

For non-DOM events (e.g., timers, Ajax responses), *EventRaceCommander* replaces each registration of an event handler h with the registration of a new event handler h' that adds h to a queue maintained by the event controller. This involves intercepting calls to a small set of global functions (e.g., `setTimeout`), and instrumenting all property assignments to intercept registrations to, e.g., the `onreadystatechange` property of `XMLHttpRequest` objects.

For DOM events (e.g., `click`, `load`), the situation is slightly more complicated due to *capturing* and *bubbling*. These event delegation mechanisms propagate events from the document root to the target node and back [1]. *EventRaceCommander* handles DOM events as follows. When the page starts loading, event handlers for all DOM event types are registered for the capturing phase of the root element (this ensures that these event handlers are triggered first, since event handlers are triggered in registration order). When one of these event handlers is invoked with an event e that was not previously postponed, the event controller is notified that e has been emitted. The controller then queries the repair policy for the action a' associated with e. If $a' \equiv \textsc{Dispatch}$, then all event handlers associated with e are triggered, and the controller is notified that e has been dispatched. Otherwise, $a' \in \{\textsc{Discard}, \textsc{Postpone}\}$, and the execution of the application's event handlers and other possible side-effects of the event (e.g., the insertion of a character into a text field) are prevented by calling `stopImmediatePropagation` and `preventDefault` on the event object of e. Furthermore, if $a' \equiv \textsc{Postpone}$, then the process is repeated by re-dispatching e asynchronously.

B. Intercepting relevant operations

EventRaceCommander intercepts *fork*, *begin* and *end* instructions. Operations of type *fork* are intercepted by replacing certain browser API functions and intercepting property assignments. For example, the `send` function on the prototype of `XMLHttpRequest` is replaced by a function that, in addition to sending the request, notifies the event controller that an Ajax response event has been forked.

It is insufficient to monitor events for which the program has an event handler: in order to enforce, e.g., $\mathcal{P}_{\mathit{async,fifo}}$, all Ajax response events must be intercepted, even those that have no response event handler. *EventRaceCommander* therefore adds a default event handler for such events.

VII. Evaluation

We aim to answer the following research questions.

RQ1: How effective is each of the application-independent policies of Section V at repairing event race errors?

RQ2: What is the impact of each application-independent repair policy on runtime performance and user experience?

RQ3: Is it possible to reduce runtime overhead and improve user experience using application-specific policies?

A. Experimental Methodology

Selecting event race errors: We use existing tools, such as, EventRacer [25] and *R*^{4} [11], to identify candidate event races in the web applications of the 20 largest companies from the Fortune 500 list. Since front pages of many websites often contain little dynamic behavior, we manually explore the selected sites to find interesting pages.

Following Mutlu et al. [20], we focus on *observable* races that result in errors, such as, uncaught exceptions or visual differences so that we can confirm the effectiveness of our repairs. In order to keep the amount of work manageable, we examine up to 25 candidate races for each website to identify whether they are observable. Altogether this gives us 117 errors that are caused by observable races.

Selecting application-independent repair policies: We study each observable race in detail to identify which of the application-independent repair policies that can repair the corresponding error.

Measuring instrumentation overhead: For each website, we create an application-independent policy that repairs all race errors (possibly by combining multiple application-independent policies), and measure the overhead of that policy. We use the Chrome Debugging Protocol [2] to measure: (i) *parsing time* (i.e., time to `DOMContentLoaded`), showing the cost for loading *EventRaceCommander* and instrumenting the source, and (ii) *layout time* (i.e., time to last layout event during initialization). In this experiment, we prevent layout events from triggering indefinitely (e.g., due to a slideshow) by stopping recursive timer registrations and intervals so that every web application terminates. We report the mean of 50 repetitions in each case.

User experience: Parsing time and layout time indirectly reflect the user's experience: most elements are ready for user interactions after a page has been parsed, and layout time reflects perceived responsiveness. In a few cases where application-independent policies are inadequate because of undesirable impact on the user experience, we attempt to design application-specific versions of application-independent policies that do no exhibit similar problems. For each such case, we attempt to evaluate the impact on user experience by comparing the delays in event processing for application-independent and application-specific policies.

System details: We run the experiments on Ubuntu 15.10 with an Intel Core i7-3770 CPU and 16 GB RAM.

Table I shows the sites and races used to evaluate *EventRaceCommander*.[2] The "Race errors" column presents the total number of observable races found in each site. The "Race classification" columns classify these races. Most of the observable races that we found are initialization races, and nearly all of these involve user events, except a race on `att.com`, where two dependent scripts are loaded without any ordering, and on `exxon.com`, where an `iframe load` event handler is registered late. Late event handler registrations tend to be a recurring problem. We also identify multiple post-initialization races. These typically cause a web application to end up in an inconsistent state.

[2]We did not detect any observable races on `berkshirehathaway.com`, `valero.com`, `unitedhealthgroup.com`, and `kroger.com`. Those sites are excluded from the table.

B. Experimental Results

RQ1: The "Repair policy" columns of Table I reflect the applicability of the application-independent policies. If, for a given site, an event race r_i appears in the column of repair policy $\mathcal{P}$, then $\mathcal{P}$ repairs the error caused by r_i. Otherwise, no application-independent policy prevents r_i, and the race appears in the "None" column. In our experiments, *all* observable races that could not be repaired using application-independent policies involve indirect user inputs (Section V-D). These races are relatively harmless (e.g., dropdowns that do not unfold when the user hovers a menu item with the cursor during loading). Note that races with the same classification tend to be prevented using the same policies. This is to be expected, since $\mathcal{P}_{\mathit{init,user}}$, $\mathcal{P}^{+}_{\mathit{init,user}}$ and $\mathcal{P}_{\mathit{init,system}}$ target initialization races, unlike $\mathcal{P}_{\mathit{async,fifo}}$ and $\mathcal{P}_{\mathit{async,user}}$.

Although we cannot guarantee that our application-independent policies always suffice, our results suggest that the policies can prevent most event race errors in practice: *94 of the 117 event race errors are repairable using our application-independent policies.*

This also suggests that, although *EventRaceCommander* relies on a light-weight instrumentation, it provides sufficient control of the nondeterminism to prevent the races that occur in practice. Furthermore, the results indicate that our assumption of what "good" schedules are (Section II-B) agrees with developers' expectations (otherwise, our policies would enforce erroneous schedules).

Table I shows that many race errors can be repaired using more than one application-independent policy. Not surprisingly, many races can be repaired using both $\mathcal{P}_{\mathit{init,user}}$ and $\mathcal{P}^{+}_{\mathit{init,user}}$, but we also find that $\mathcal{P}_{\mathit{async,fifo}}$ and $\mathcal{P}_{\mathit{async,user}}$ often repair the same race. This happens when a user triggers an asynchronous event (e.g., an Ajax request) twice. The policy $\mathcal{P}_{\mathit{async,fifo}}$ avoids such races by enforcing an order among the unordered events, whereas $\mathcal{P}_{\mathit{async,user}}$ postpones user events while an asynchronous event is pending (thereby ensuring that event handlers and their forked events execute atomically).

RQ2: The last two columns of Table I show parsing and layout time. For most sites, the instrumentation overhead is less than 200ms, which we deem to be acceptable. Small websites exhibit larger relative overheads due the cost of including *EventRaceCommander*'s 32 KB of JavaScript. The absolute overhead is barely noticeable by a user, though.

Regarding user experience, it is important to interrupt only user events that are involved in races, and only for as long as is necessary to prevent undesirable schedules. Generally, we find that the policies $\mathcal{P}_{\mathit{init,system}}$ and $\mathcal{P}_{\mathit{async,fifo}}$ can be enforced obliviously to the user, since they do not involve user events and, in our experiments, do not significantly postpone UI updates. There is often room for improvements over $\mathcal{P}_{\mathit{init,user}}$, $\mathcal{P}_{\mathit{async,user}}$, and $\mathcal{P}^{+}_{\mathit{init,user}}$, since the operation predicates in these policies are overly general. This is mostly a problem for $\mathcal{P}^{+}_{\mathit{init,user}}$ in sites that extensively load code asynchronously (e.g., `walmart.com`, which uses RequireJS [4]). In such cases, the page appears to be ready significantly before user input is tolerated, and an application-specific policy should be used

		Race classification				**Repair policy**						**Instrumentation overhead**	
		Initialization races			*Post-init. races*								
Website	Race errors	declare/ event	register/ event	update/ event	system/ user	$\mathcal{P}_{init,user}$	$\mathcal{P}^{+}_{init,user}$	$\mathcal{P}_{init,system}$	$\mathcal{P}_{async,fifo}$	$\mathcal{P}_{async,user}$	None	Parsing (ms)	Layout (ms)
`walmart.com`	14		$r_1 \ldots r_{13}$		r_{14}		$r_1 \ldots r_{10}$		r_{14}	r_{14}	$r_{11} \ldots r_{13}$	+609 (1.29x)	+247 (1.08x)
`exxon.com`	7		$r_1 \ldots r_6$		r_7	$r_1 \ldots r_5$	$r_1 \ldots r_5$	r_6	r_7	r_7		+20 (1.02x)	+23 (1.01x)
`chevron.com`	8		$r_1 \ldots r_6$		$r_7 \ldots r_8$				$r_7 \ldots r_8$	$r_7 \ldots r_8$	$r_1 \ldots r_6$	+88 (1.12x)	+176 (1.13x)
`apple.com`	3		$r_1 \ldots r_2$		r_3	$r_1 \ldots r_2$	$r_1 \ldots r_2$		r_3	r_3		+69 (1.11x)	+65 (1.10x)
`gm.com`	8		$r_1 \ldots r_6$	r_7	r_8	$r_1 \ldots r_6$	$r_1 \ldots r_7$		r_8	r_8		+60 (1.08x)	+60 (1.08x)
`phillips66.com`	3	$r_1 \ldots r_2$			r_3	r_1	$r_1 \ldots r_2$		r_3	r_3		+87 (1.14x)	+31 (1.04x)
`ge.com`	10			$r_1 \ldots r_7$	$r_8 \ldots r_{10}$	r_1	$r_2 \ldots r_7$		$r_8 \ldots r_{10}$	$r_8 \ldots r_{10}$		+124 (1.08x)	+207 (1.13x)
`ford.com`	1		r_1			r_1	r_1					+154 (1.06x)	+155 (1.06x)
`cvshealth.com`	10		$r_1 \ldots r_8$		$r_9 \ldots r_{10}$	$r_1 \ldots r_7$	$r_1 \ldots r_7$		$r_9 \ldots r_{10}$	$r_9 \ldots r_{10}$	r_8	-24 (0.98x)	-14 (0.99x)
`mckesson.com`	2	r_1		r_2		$r_1 \ldots r_2$	$r_1 \ldots r_2$					+101 (1.08x)	+7 (1.00x)
`att.com`	12	$r_1 \ldots r_2$	$r_3 \ldots r_{12}$			$r_3 \ldots r_{12}$	$r_3 \ldots r_{12}$		$r_1 \ldots r_2$			+723 (1.21x)	+699 (1.20x)
`verizonwireless.com`	13	$r_1 \ldots r_{13}$				$r_1 \ldots r_{13}$	$r_1 \ldots r_{13}$					+360 (1.15x)	+266 (1.12)
`amerisourcebergen.com`	4		$r_1 \ldots r_4$			$r_1 \ldots r_4$	$r_1 \ldots r_4$					+13 (1.04x)	+12 (1.03x)
`fanniemae.com`	5		$r_1 \ldots r_5$			$r_1 \ldots r_5$	$r_1 \ldots r_5$					+143 (1.26x)	+70 (1.10x)
`costco.com`	16		$r_1 \ldots r_{16}$			$r_1 \ldots r_3$	$r_1 \ldots r_3$				$r_4 \ldots r_{16}$	+92 (1.11x)	+45 (1.03x)
`hp.com`	1		r_1			r_1	r_1					+35 (1.01x)	+37 (1.01x)
total	117	18	78	9	12	61	78	1	14	12	23		

TABLE I
OBSERVABLE RACES, APPLICABILITY OF APPLICATION-INDEPENDENT REPAIR POLICIES, AND INSTRUMENTATION OVERHEAD.

Race classification: *declare/event*: an entity may be used before it is declared, triggering an error (e.g., scenario Ⓑ). *register/event*: an event handler may be registered late, leading to lost events (e.g., scenario Ⓐ, lines 47–51). *update/event*: a form field may be updated asynchronously, overwriting the user's input (e.g., lines 54–63). *system/user*: a system and user event are unordered, leading to an error or erroneous state (e.g., scenario Ⓒ).

to target the relevant user events, and minimize the time in which the user is disrupted.

Interestingly, we find that some of the websites, e.g. `apple.com`, prevent races in ways similar to $\mathcal{P}_{async,user}$, by showing a spinner that takes up most space on the screen, when a user event leads to asynchronous DOM updates.

RQ3: We now briefly report on two event races where application-independent repair policies yield suboptimal results, and discuss how each situation can be remedied using an application-specific policy.

On `att.com`, event race r_1 can cause a `TypeError` due to two scripts being unordered.[3] Policy $\mathcal{P}_{async,fifo}$ ensures that asynchronous scripts are executed in FIFO order and fixes the error, but unnecessarily imposes an order on 39 scripts. On average, 21 of these scripts are postponed for 292ms. This can be prevented using a specialized policy $\mathcal{P}'_{async,fifo}$, which only postpones the execution of `satellite-567046aa64746d0712008241.js`. On average, this policy postpones no scripts at all (i.e., in our experiments, the two scripts always load in the desired order).

On `walmart.com`, a `click` event handler of a button is registered by an asynchronous script. Until that happens, `click` events on the button are lost and no dropdown is shown (event race error r_{13}). While this problem can be fixed using the application-independent policy $\mathcal{P}^{+}_{init,user}$, this results in excessive delays for processing a `click` event. We can avoid such undesirable impact on the user experience by designing an application-specific policy $\mathcal{P}_{spec}$ that postpones `click` events only until the handler is present. In an experiment, we issue a click immediately when the button is declared, and measure the time until the corresponding event handlers execute. On average, the `click` event is dispatched 817ms faster when using the policy $\mathcal{P}_{spec}$ instead of $\mathcal{P}^{+}_{init,user}$.

The application-specific repair policies discussed above are both "optimal" in the sense that they only postpone events that are involved in the races under consideration, for the minimal amount of time required to prevent the undesired orders. We argue that, for these race errors, enforcing repair policies using *EventRaceCommander* compares well to alternative solutions such as modifying the code to introduce ad-hoc synchronization or explicitly load scripts synchronously.

C. Discussion

Some aspects of our evaluation may affect the generality of the reported results. Most significantly, the selection of websites and event race errors used in our evaluation could be subject to bias. We have attempted to address this concern by evaluating *EventRaceCommander* on the websites of the 20 largest companies from the Fortune 500 list, similar to previous work on event race detection [11, 25].

The code of the websites used in our evaluation may be subject to change, which may affect reproducibility of our results. Therefore, we spent significant effort using *mitmproxy* [3] to record the server responses for an interaction with every site under consideration. This enables reproducibility for all front pages. Regrettably, some highly dynamic pages that we consider cannot be replayed, since the URLs of Ajax requests depend on user input, random numbers, timestamps, etc. Still, this is a significant improvement over recent work [8, 10, 11, 21, 22, 25, 28, 34], where the importance of reproducibility has mostly been ignored. The recordings from our study are available with the implementation of *EventRaceCommander*.

[3]The global variable `s_att` is declared in `s-code-contents-65778bc202aa3fe01113e6b6ea6d103eda099fe5.js`, and used in `satellite-567046aa64746d0712008241.js`. The latter may, depending on the event order, crash during an assignment to `s_att.events`.

A related concern is that real websites may give rise to unpredictable network delays, which may affect repair policies, such as, $\mathcal{P}_{async,fifo}$. In principle, these delays can become arbitrarily large, so the data from our experiments may not truly reflect the impact on user experience. In our experiments, we avoid large fluctuations by relying on recordings of every website, and by conducting experiments 50 times and reporting average times. To prevent situations where the user is being disrupted for too long, it would be possible to monitor if *EventRaceCommander* postpones an event for more than a given threshold. In such cases, the event could simply be dispatched, and the users of *EventRaceCommander* could be notified of the incident, so that the policy can be adjusted.

VIII. Related work

Race detection: Ide et al. [10] pointed out that JavaScript programs can have data races despite being single-threaded and non-preemptive. Such races often arise due to asynchronous Ajax communication and HTML parsing. The authors note regarding Ajax races that "the programmer prefers to think of the interaction with the server as a synchronous call", which is also the foundation for our scheduling policies for such races. Zheng et al. [34] proposed a static analysis for automatically detecting JavaScript races. Due to the dynamic nature of JavaScript, such static analyses are often prohibitively imprecise or unscalable. Inspired by successful techniques developed for multi-threaded programs [7], WebRacer and EventRacer instead use dynamic analysis and a happens-before relation [22, 25]. This significantly improves precision, however, these tools cannot distinguish harmful from benign races, which has motivated techniques that explore whether races cause observable differences [8, 11, 21]. Still, these techniques tend to report many false positives and also miss harmful races, and it has been observed that the harmful races that are detected are often difficult to fix.

Event race detection algorithms have also been developed for Android, using similar techniques as those targeting JavaScript, but with more sophisticated happens-before relations [5, 9, 19]. Adapting our technique to Android is an interesting opportunity for future work.

Automated fixing of race errors: The idea of automatically fixing race errors has been studied extensively in a multi-threaded setting, but not much for event-driven applications, in particular JavaScript.

Some techniques patch the program code by inserting, e.g., locks and wait-signal operations, based on reports from race detectors and static analysis [12–16, 29]. The JavaScript platform provides no explicit synchronization primitives, but our repair policy mechanism can simulate the effect of having wait-signal primitives or atomic groups of event handlers.

Other techniques steer away from nondeterministic errors by postponing selected actions, much like our approach but for multi-threaded programs. The AI technique [32] attempts to stall threads where manifestation of a concurrency bug is about to become deterministic. Kivati [6] uses static analysis and hardware watchpoints to detect atomicity violations and then dynamically reorders the relevant instructions. The Aviso system [17] learns schedule constraints from successful and failing executions, and then uses these constraints to guide scheduling, much like our policy mechanism and controller. The Loom system [30] uses a notion of execution filters, which resembles our use of application-specific repair policies.

These techniques share the limitation of *EventRaceCommander* that they cannot fix all race errors while entirely avoiding situations where actions are postponed excessively.

Other approaches include rollback-recovery [33], replicated execution with different schedules [27], replication of shared state in critical sections [23, 24], or require special hardware [18], which would not be realistic for JavaScript.

EventHealer [26], unlike most of the work mentioned above, considers event-driven programs, but with a different execution model than the one in JavaScript: execution takes place in a main thread, which has lower priority than event handlers, and preemption is possible but can be selectively disabled to protect critical sections. The system uses static analysis to locate event handlers, shared variables, and fragments of code that should be treated as critical sections, which is very different from our setting.

None of the work on automated fixing discussed above targets JavaScript. A position paper by Mutlu et al. [20] proposes a notion of "schedule shepherding" for JavaScript, but does not present any mechanism for how to actually do it. The recent ARROW tool by Wang et al. [28] is the first to automatically repair races in JavaScript applications. The key difference to *EventRaceCommander* is that ARROW is based on static analysis, which is notoriously difficult for real-world JavaScript code. Moreover, the main idea in ARROW is to identify inconsistencies between the happens-before and def-use relations, which may miss many race errors, even if more powerful static analysis were developed. ARROW cannot repair any of the errors in the example application in Section II.

IX. Conclusion

We have presented a general framework for controlling nondeterminism in event-driven applications using specified repair policies, and proposed application-independent policies to prevent nondeterminism that commonly triggers event race errors. The framework is sufficiently general to repair a wide variety of real-world event race errors. Our experimental results show that 94 of 117 event race errors are repairable by our application-independent policies, and that application-specific policies are useful to target specific races, when needed.

For future work, it will be interesting to automate the process of inferring application-specific policies for a given event race, to avoid negative impacts from overly general policies. Such candidate policies should restrict the nondeterminism only as needed to repair a given race, but still be reasonably general, so that they do not only apply to the concrete execution explored by the dynamic race detector.

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] W3C Document Object Model Level 2 Events Specification. `http://www.w3.org/TR/DOM-Level-2-Events/events.html#Events-flow`, last accessed on 2016/08/24.

[2] Chrome Debugging Protocol. `https://developer.chrome.com/devtools/docs/debugger-protocol`, last accessed on 2016/08/24.

[3] mitmproxy. `https://mitmproxy.org/`, last accessed on 2016/08/24.

[4] RequireJS. `http://requirejs.org/`, last accessed on 2016/08/24.

[5] P. Bielik, V. Raychev, and M. T. Vechev. Scalable race detection for Android applications. In *Proc. 30th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

[6] L. Chew and D. Lie. Kivati: fast detection and prevention of atomicity violations. In *Proc. 5th European Conference on Computer Systems (EuroSys)*, 2010.

[7] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. *Commun. ACM*, 53(11), 2010.

[8] S. Hong, Y. Park, and M. Kim. Detecting concurrency errors in client-side Java Script web applications. In *Proc. 7th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2014.

[9] C. Hsiao, C. Pereira, J. Yu, G. Pokam, S. Narayanasamy, P. M. Chen, Z. Kong, and J. Flinn. Race detection for event-driven mobile applications. In *Proc. 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[10] J. Ide, R. Bodik, and D. Kimelman. Concurrency concerns in rich internet applications. In *Proc. Workshop on Exploiting Concurrency Efficiently and Correctly*, 2009.

[11] C. S. Jensen, A. Møller, V. Raychev, D. Dimitrov, and M. T. Vechev. Stateless model checking of event-driven applications. In *Proc. 30th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

[12] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[13] G. Jin, W. Zhang, and D. Deng. Automated concurrency-bug fixing. In *Proc. 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[14] S. Khoshnood, M. Kusano, and C. Wang. ConcBugAssist: constraint solving for diagnosis and repair of concurrency bugs. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 2015.

[15] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing data races on-the-fly. In *Proc. 5th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 2007.

[16] P. Liu and C. Zhang. Axis: Automatically fixing atomicity violations through solving control constraints. In *Proc. 34th International Conference on Software Engineering (ICSE)*, 2012.

[17] B. Lucia and L. Ceze. Cooperative empirical failure avoidance for multithreaded programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[18] B. Lucia, J. Devietti, L. Ceze, and K. Strauss. Atom-Aid: Detecting and surviving atomicity violations. *IEEE Micro*, 29(1), 2009.

[19] P. Maiya, A. Kanade, and R. Majumdar. Race detection for Android applications. In *Proc. 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[20] E. Mutlu, S. Tasiran, and B. Livshits. I know it when I see it: Observable races in JavaScript applications. In *Proc. Workshop on Dynamic Languages and Applications (Dyla)*, 2014.

[21] E. Mutlu, S. Tasiran, and B. Livshits. Detecting JavaScript races that matter. In *Proc. 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015.

[22] B. Petrov, M. T. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *Proc. 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[23] S. K. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani. ISOLATOR: dynamically ensuring isolation in concurrent programs. In *Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[24] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. G. Zorn, R. Nagpal, and K. Pattabiraman. Efficient runtime detection and toleration of asymmetric races. *IEEE Trans. Computers*, 61(4), 2012.

[25] V. Raychev, M. T. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *Proc. 28th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages, and Applications (OOPSLA)*, 2013.

[26] G. M. Tchamgoue, K. H. Kim, and Y. Jun. EventHealer: Bypassing data races in event-driven programs. *Journal of Systems and Software*, 118, 2016.

[27] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[28] W. Wang, Y. Zheng, P. Liu, L. Xu, X. Zhang, and P. Eugster. ARROW: Automated repair of races on client-side web pages. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 2016.

[29] D. Weeratunge, X. Zhang, and S. Jagannathan. Accentuating the positive: atomicity inference and enforcement using correct executions. In *Proc. 26th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[30] J. Wu, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. In *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[31] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. N. Bairavasundaram. How do fixes become bugs? In *Proc. 13th European Software Engineering Conference and 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2011.

[32] M. Zhang, Y. Wu, S. Lu, S. Qi, J. Ren, and W. Zheng. AI: a lightweight system for tolerating concurrency bugs. In *Proc. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[33] W. Zhang, M. de Kruijf, A. Li, S. Lu, and K. Sankaralingam. ConAir: featherweight concurrency bug recovery via single-threaded idempotent execution. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[34] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proc. 20th International Conference on World Wide Web (WWW)*, 2011.