RClassify: Classifying Race Conditions in Web Applications via Deterministic Replay

Lu Zhang
Virginia Tech
Blacksburg, VA, USA

Chao Wang
University of Southern California
Los Angeles, CA, USA

***Abstract*—Race conditions are common in web applications but are difficult to diagnose and repair. Although there exist tools for detecting races in web applications, they all report a large number of false positives. That is, the races they report are either *bogus*, meaning they can never occur in practice, or *benign*, meaning they do not lead to erroneous behaviors. Since manually diagnosing them is tedious and error prone, reporting these race warnings to developers would be counter-productive. We propose a platform-agnostic, deterministic replay-based method for identifying not only the *real* but also the *truly harmful* race conditions. It relies on executing each pair of racing events in two different orders and assessing their impact on the program state: we say a race is harmful only if (1) both of the two executions are feasible and (2) they lead to different program states. We have evaluated our *evidence-based* classification method on a large set of real websites from Fortune-500 companies and demonstrated that it significantly outperforms all state-of-the-art techniques.**

I. Introduction

Modern web applications are complex due to their need to implement many features on the client side through asynchronous programming and the use of JavaScript code while maintaining quick response to users. Although web browsers typically guarantee that each JavaScript code block is executed atomically, meaning there is no *data-race* in the traditional sense, high-level *race conditions* can still occur due to deferred HTML parsing, interleaved execution of event handlers, timers, Ajax requests, and their callbacks.

Existing race detection tools for web applications [33], [21], [22], [7], [18], [9] often report many false positives. That is, warnings reported by these tools may be *bogus*, or *real* but *harmless*. For example, EVENTRACER [22] reported hundreds of warnings from the official websites of Fortune-500 companies; although some of these race conditions are indeed harmful, the vast majority are not, which means directly reporting them to developers would have been counter-productive. None of the existing tools, including R4 [9], can accurately assess the impact of racing events and robustly identify the real and truly harmful race conditions.

We propose RCLASSIFY, the first *evidence-based* method for classifying race-condition warnings in web applications. Toward this end, we develop a platform-agnostic deterministic replay framework for client-side JavaScript programs, and leverage it to assess the actual impact of race conditions. Given a race-condition warning denoted (ev_a, ev_b), where ev_a and ev_b are the racing events, we first execute the application while forcing ev_a to occur before ev_b, and then execute the application while forcing ev_b to occur before ev_a. We say that (ev_a, ev_b) is a harmful race only if (1) both executions are feasible and (2) the resulting program states, ps_1 and ps_2, differ in some important fields of the HTML DOM, JavaScript variables, and environment variables of the browser. The intuition is that, when the order of ev_a and ev_b is not fully controlled by the program logic, and yet affects the program behavior, it deserves a closer look by developers.

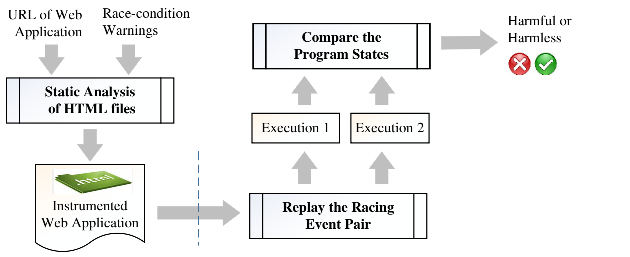
URL of Web Application
Race-condition Warnings
Static Analysis of HTML files
Instrumented Web Application
Replay the Racing Event Pair
Execution 1
Execution 2
Compare the Program States
Harmful or Harmless

Fig. 1. RCLASSIFY: Our *evidence-based* race-condition classification method.

The overall flow of RCLASSIFY is shown in Fig. 1, whose input is the URL of the web application and a set of race-condition warnings, and whose output is the set of harmful races. First, it statically analyzes the HTML files and then uses source-code transformation to add self-monitoring and control capabilities. Then, it analyzes the race-condition warnings (reported by the race detection tool) and generates the configuration files needed for deterministic replay. Next, for each pair (ev_a, ev_b) of racing events, it executes the application twice—once with ev_a preceding ev_b and the other time with ev_b preceding ev_a. Finally, it compares the two resulting program states ps_1 and ps_2.

There are two technical challenges. The first challenge is developing a robust method to deterministically replay a JavaScript-based client-side web application. This is difficult due to the myriad possible sources of nondeterminism. For example, the race condition may occur during the deferred parsing of HTML elements, the interleaved execution of JavaScript, the dispatch of multiple event handlers, the firing of timer events, the execution of asynchronous HTTP requests, as well as their callback routines. In this context, our main contribution is developing a unified framework for controlling the execution order of the various types of racing events.

Our replay framework differs from the mechanisms used by existing race detection tools such as EVENTRACER, WAVE, and R4, because it is implemented within the target web application itself and therefore is platform-agnostic. That is, we do not modify the web browsers or their underlying JavaScript execution engines (e.g., WEBKIT). Instead, we leverage source-code transformation to add self-monitoring and control capabilities to the application itself (see Section V). This is better than existing approaches because technologies are changing rapidly and tools implemented using a particular version of the browser will quickly become obsolete. In contrast, our platform-agnostic approach will be more robust against these changes and updates.

Since we concretely execute the application using deterministic replay, as opposed to heuristically filtering the warnings [21], [22] or applying conservative static analysis [18], [32], we can robustly decide if a race condition is real (i.e., if both execution orders are feasible). The reason why existing tools report many bogus race conditions in the first place is because some hidden happens-before relations between events are not accounted for, and precisely capturing all happens-before relations would have been prohibitively expensive.

The second challenge is to decide, during state recording and comparison, which fields of the program state are important and thus should be compared. For a typical client-side web application, the number of fields can be extremely large, which means including all of them would result in large overhead at run time. Furthermore, many fields are actually designed to be sensitive to other sources of nondeterminism that are irrelevant to the race condition. For example, there are fields that need to have different values depending on the date or time of the day. In such cases, we should exclude them in order to avoid the false positives.

Therefore, our main contribution in this context is developing a flexible configuration interface to allow users to specify which fields should be excluded. We also propose a testing-based method (see Section VI) to automatically identify and exclude these irrelevant fields.

We implemented and evaluated RClassify on standard benchmarks and real websites from Fortune-500 companies. Our experiments show that RClassify outperforms all other existing tools capable of handling the same benchmarks, including EventRacer [22], Mutlu et al. [18], and R4 [9]. For example, RClassify identified all 33 known-to-be-harmful races out of the 50 warnings in standard benchmarks, whereas R4 identified only 8 of them and Mutlu et al. [18] did not identify any. Furthermore, on the seventy randomly chosen websites from the portals of Fortunate-500 companies, EventRacer [22] returned 1,903 warnings, among which RClassify identified 73 as bogus, 132 as harmful, 1644 as harmless, and 54 as undecided. We manually reviewed the 132 harmful races and confirmed the correctness of our classification; in contrast, R4 [9] identified only 33 of the harmful races, indicating that it is significantly less effective.

To sum up, this paper makes the following contributions:

- We propose an *evidence-based* method for classifying race-condition warnings in web applications, by concretely executing the application to assess the actual impact of racing events.
- We develop a *platform-agnostic* deterministic replay framework for JavaScript-based web applications, which does not rely on modifying browsers or JavaScript engines, and thus is more widely applicable.
- We evaluate the new method on standard benchmarks as well as a large number of real websites to demonstrate its effectiveness in identifying the harmful race conditions.

```
<html>
 <head> ... </head>
 <body>
  <img src="image1.jpg" onload="image1Loaded()"
        id="image1">
  <!-- omitted elements... -->
  <script id="script1">
    function image1Loaded() {
      document.getElementById("button1")
            .addEventListener("click", func);
    }
    function func() {
      document.getElementById("outputField").innerHTML
            = "Well done!";
    }
  </script>
  <!-- omitted elements... -->
  <button id="button1"> button1 </button>
  <!-- omitted elements... -->
  <div id="outputField"> </div>
 </body>
</html>
```

Fig. 2. Example: A client-side web application with race conditions.

II. Motivation

In this section, we use examples to illustrate the ideas behind our new method while highlighting the technical challenges.

A. Race Conditions

Consider the web page in Fig. 2, which contains an image, a button, and a JavaScript code block. The *image.onload* event, fired after the browser downloads `image1`, invokes the `image1Loaded()` function, which in turn attaches a listener function to the *onclick* event of `button1`. The button may be clicked by the user immediately after it is parsed, and its listener function `func()` changes text in `outputField` to `'Well done!'` Thus, the expected event sequence is ev_0: *parsing(*`image1`*)* $\rightarrow$ ev_1: *parsing(*`script1`*)* $\rightarrow$ ev_2: *parsing(*`button1`*)* $\rightarrow$ ev_3: *firing(*`image1`*.onload)* $\rightarrow$ ev_4: *parsing (*`outputField`*)* $\rightarrow$ ev_5: *firing(*`button1`*.onclick)*.

However, depending on the network speed, load of the computer, and timing of the user click, there may be other execution sequences, some of which do not lead to the expected display of `'Well done!'`. As shown in the partial order of events in Fig. 3, there are four race conditions:

1) **RC1** is (ev_1, ev_3) over `image1Loaded`
   a) event ev_1: *parsing(*`script1`*)*
   b) event ev_3: *firing(*`image1`*.onload)*
2) **RC2** is (ev_2, ev_3) over `button1`
   a) event ev_2: *parsing(*`button1`*)*
   b) event ev_3: *firing(*`image1`*.onload)*
3) **RC3** is (ev_3, ev_5) over `button1`
   a) event ev_3: *firing(*`image1`*.onload)*
   b) event ev_5: *firing(*`button1`*.onclick)*
4) **RC4** is (ev_4, ev_5) over `outputField`
   a) event ev_4: *parsing(*`outputField`*)*
   b) event ev_5: *firing(*`button1`*.onclick)*

The first race condition (RC1) is between the parsing of HTML element `script1` and the firing of *image1.onload*. Typically, the parsing finishes first, but if `image1` is downloaded before the parsing finishes, e.g., due to caching of the image or slow parsing of other HTML elements preceding `script1`,

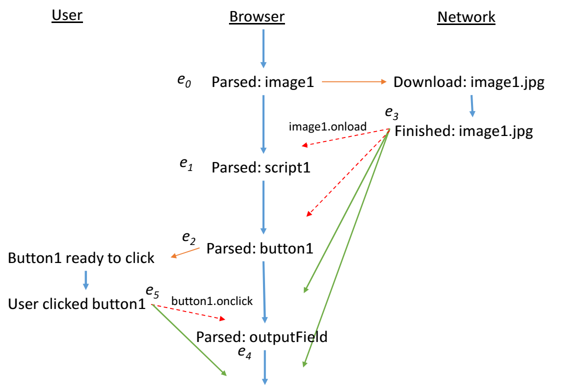

Fig. 3. Partial order graph of the racing events of the example in Fig. 2.

```
<input id="input_1" type="text" value="Name"
      onfocus="clearText(this)">
<input id="input_2" type="text" value="Email"
      onfocus="clearText(this)">
<script id="script2" src="nyl-min.js"> ... </script>
<script id="script1">
  <!-- adding async script "utag.js" to DOM -->
</script>
...
<!-- nyl-min.js -->
function clearText(a){
  if(a.defaultValue==a.value) {a.value=""} }
...
<!-- utag.js -->
function c() {
  if(!done) { done = true; ... } }
document.addEventListener("DOMContentLoaded",c,false);
window.addEventListener("load", c, false);
```

Fig. 4. Race conditions that we detected from *www.newyorklife.com*.

`image1Loaded` will be undefined when the browser invokes it through *`image1`.onload*.

The second race condition (RC2) is between the parsing of `button1` and the firing of *`image1`.onload*. Here, we assume the unwanted situation regarding RC1 did not occur (ev_1 precedes ev_3 and `image1Loaded` is properly defined). In this case, it is still possible for *`image1`.onload* to happen before the parsing of `button1`, thereby causing *document.getElementById (“button1”)* at Line 8 to fail.

The third race condition (RC3) is between *`image1`.onload* and *`button1`.onclick*. Here, we again assume the unwanted situations regarding RC1 and RC2 did not occur (ev_1 and ev_2 precede ev_3). However, it is possible for `button1` to be clicked before the firing of *`image1`.onload*. Since `func()` has not yet been attached to *`button1`.onclick*, clicking the button will not lead to the desired behavior.

The fourth race condition (RC4) is between the parsing of HTML element `outputField` and the firing of *`button1`.onclick*. Here, we again assume that none of the unwanted situations regarding RC1, RC2, and RC3 occurred. However, it is possible for `button1` to be clicked and yet the `'Well done!'` message is not displayed. This occurs when the button is clicked before the parsing of `outputField`, causing *document.getElementById (“outputField”)* at Line 11 to fail.

The example in Fig. 2 and Fig. 3 shows that race conditions in web applications are tricky: Developers may have to spend long hours weeding out the bogus warnings, if the race detection tools report too many false positives to them. Our work aims to lift this burden from developers, thus allowing them to focus on diagnosing the truly harmful race conditions.

B. Harmful Race Conditions

We leverage deterministic replay and state comparison to identify harmful races. Let (ev_a, ev_b) be a potential race condition (e.g., reported by EVENTRACER). We perform controlled executions of the application twice, first with ev_a before ev_b, and then with ev_a after ev_b. If any of these executions is infeasible, the replay fails, indicating that it is a bogus warning. Otherwise, we record the program states at the end of the two executions and compare them.

Consider RC1 in Fig. 2. First, we execute the application while forcing ev_1 before ev_3, which leads to the expected event sequence. Then, we execute the application while forcing ev_3 before ev_1. Both executions are feasible, and the second execution invokes `image1Loaded` before it is defined, causing the browser to print an error message in the console window. The event handler for *`button1`.onclick* also remains uninitialized, subsequently causing a difference in `outputField`. Therefore, we say that RC1 is a *harmful* race, meaning that it deserves the attention of the developers.

However, not all *real* race conditions are harmful. Among the three race conditions in Fig. 4, which were detected by RCLASSIFY from *www.newyorklife.com*, the one between parsing of *utag.js* and firing of *document.onDOMContentLoaded* is harmless, while the two races over `input_1` and `input_2` are harmful. The first race is harmless because although the event handler of *document.onDOMContentLoaded* may be registered (by *utag.js*) after the browser fires the *document.onDOMContentLoaded* event, the same callback function `c()` is also registered to *window.onload* as a precaution. Since *window.onload* always execute after the load of *utag.js*, it ensures that `c()` is executed, thereby resulting in the same program state.

However, the two races over `input_1` and `input_2` are harmful because they initially show the hint values `'Name'` and `'Email'` to the user, but their *onfocus* event handlers, which call `clearText()`, will empty these hint values as soon as the user tries to type into the text areas. Unfortunately, the script that defines `clearText()` may be parsed after the user typed something into the two fields – assume `'USERTYPED'` is what the user typed. This can lead to unwanted text contents such as `'NamUSERTYPEDe'` or `'NameUSERTYPED'` instead of just `'USERTYPED'`. Our *evidence-based* method can correctly identify these two races as harmful, since they led to significant differences in the program states. In contrast, EVENTRACER [22] reported all three races to the user, whereas Mutlu et al. [18] could not detect any of them.

III. PRELIMINARIES

A. Web Applications

A modern HTML page consists of a set of elements, each with opening/closing tags and the content in between, e.g., `<p>...</p>` for a paragraph. In addition, elements may be added dynamically by executing JavaScript code. The document object model (DOM) is a tree representation of the web page to be rendered by the browser. Each DOM node has attributes for holding meta-data, e.g., the `<img>` element has the `src` attribute indicating the URL of the image. JavaScript may be embedded in the HTML file or declared externally, e.g., `<script src="code.js"> </script>`; by default, they are synchronous and therefore must be parsed before the browser can move to the next HTML element. However, external scripts with the "defer" attribute will run after all static HTML elements are parsed, whereas "async" scripts may run at any time after they are downloaded.

Web applications follow an event-driven execution model, where various handlers are registered to events of DOM nodes to react to the user and the environment. An event may be propagated through the DOM tree through "capturing" and "bubbling". For example, if a button is clicked, the *onclick* event will be fired not only for this button but also for DOM nodes that recursively contain this button. The propagation is performed level by level, all the way up to the *window* object. Any *onclick* event handler registered to DOM nodes along this chain of propagation will also be fired.

The execution trace of a web application is a sequence of events, denoted $\rho = ev_1, \ldots, ev_n$. Each event is of the form $ev = \langle id, type, info, mem \rangle$, where id is the element ID, $type$ is the event type, $info$ stores additional information of the DOM node, and mem stores information of shared memory access. We consider five event types:

- $parse(element)$, which represents the parsing of an element. Static elements are parsed sequentially following the order in which they are declared in the HTML.
- $execute(js)$, which represents the execution of an embedded, deferred, or asynchronous JavaScript code block.
- $fire(ev\_cb)$, which represents the execution of a callback function ev_cb in response of an event, such as the *onclick* event of a button.
- $fire(tm\_cb)$, which represents the execution of a callback function tm_cb in response of a timer, registered using either *setInterval()* or *setTimeOut()*.
- $fire(ajax\_cb)$, which represents the execution of a callback function $ajax\_cb$ in response to an Ajax request.

Let PS be the set of program states. Each state $ps = \langle DOM, JS, Env, Console \rangle$ is a memory snapshot, where DOM is the DOM content, JS is a valuation of JavaScript variables, Env is a valuation of the browser's environment variables, and $Console$ denotes console messages. An execution is a sequence $\lambda = ps_0 \xrightarrow{ev_1} ps_1 \xrightarrow{ev_2} ps_2 ... \xrightarrow{ev_n} ps_n$, where $ps_1, \ldots, ps_n \in PS$ are program states, $ev_1, \ldots, ev_n$ are events, and for each $1 \leq i < n$, we have $ps_{i-1} \xrightarrow{ev_i} ps_i$.

B. Race Conditions

Since the web browser ensures that each JavaScript-based event handler function is executed atomically, it is impossible for individual JavaScript statements to interleave. Thus, web applications do not have *data-races* in the traditional sense (unlike multithreaded Java or C++ programs). However, there are still *race conditions* at higher levels. For example, a web application may consider fast rendering of visual elements as a priority, while asynchronously loading images and JavaScript libraries. In such cases, event handlers of the visual elements may be made available to users long before the corresponding JavaScript functions are defined.

Before formally defining race conditions, we define the must-happens-before relation $\rightarrow_{mhb}$, which is a binary relation over events. We say $(ev_a, ev_b) \in \rightarrow_{mhb}$ if and only if ev_a precedes ev_b in all possible executions of the web application. If ev_a precedes ev_b only in a particular execution, we say $ev_a \prec ev_b$ in this execution. Let $WR(ev)$ be the set of memory locations written by ev, and $RD(ev)$ be the set of memory locations read by ev. A race condition is defined as a pair (ev_a, ev_b) of events such that

1) $(ev_a, ev_b) \notin \rightarrow_{mhb}$, $(ev_b, ev_a) \notin \rightarrow_{mhb}$, and
2) $\exists\ var$ such that $var \in WR(ev_a) \cap WR(ev_b)$ or $var \in WR(ev_a) \cap RD(ev_b)$ or $var \in RD(ev_a) \cap WR(ev_b)$.

However, (ev_a, ev_b) may not be considered harmful if it does not affect the program behavior [1]. Thus, we say a race condition is harmful only if there exist two executions λ_1 and λ_2 such that (1) $ev_a \prec ev_b$ in λ_1, (2) $ev_b \prec ev_a$ in λ_2, and (3) the two resulting program states ps_1 and ps_2 are different.

IV. The Algorithm of RClassify

RClassify takes the URL of the web application and a set of race-condition warnings as input, and returns the classification results as output (Fig. 1). It first removes the bogus races, where at least one of the two executions is shown to be infeasible. Next, it removes the real but harmless races, where the two executions do not result in differences in the program states. In both steps, we need to deterministically replay the web application.

Toward this end, we first download the application following the URL and then instrument the HTML files to add self-logging capabilities. We want to record information of the racing events during race detection, so we can identify and control these events during deterministic replay. In this work, we used EventRacer [22] to generate race-condition warnings (input of RClassify), although other race detection tools may be used as well.

Next, we classify these warnings using replay and state comparison. For each pair (ev_a, ev_b) of racing events, we first execute ev_a before ev_b (denote λ_1) and then execute ev_b before ev_a (denoted λ_2). In both executions, we fix the order of other events as much as possible.

If both executions are feasible, we call (ev_a, ev_b) a real race condition, and record the global states at the end. Let ps_1 and ps_2 be the two states resulting from λ_1 and λ_2, respectively. We say (ev_a, ev_b) is a harmful race if there are significant differences in ps_1 and ps_2.

In the remainder of this section, we explain how to instrument the web application and analyze the race-condition warnings to prepare for the subsequent deterministic replay. In Sections V and VI, we will explain how to control the event order and record/compare program states.

A. Instrumenting the HTML Files

One input of RClassify is the URL of the target web application, consisting of the HTML files and other resource files, such as Cascading Style Sheets (CSS), JavaScript code, images, audio, and video. After fetching these files, we instrument them before using them as input of the race detection tool (EventRacer); it allows us to generate information of the racing events for deterministic replay.

```
1  <html>
2   <head>
3     <script src="Pre_RClassify.js"></script>
4     <!-- start of head elements -->
5     ...
6    </head>
7    <body>
8     ...
9     <!-- end of body elements -->
10    <script> window.addEventListener("load",
            RC_fire_handlers, false); </script>
11   </body>
12 </html>
```

Fig. 5. Example: Instrumented web page prior to race detection.

We use an open-source HTML parser called JSOUP (http://jsoup.org/). We generate a unique *id* attribute for each HTML element, to help pinpoint the HTML element involved in the racing event at run time and therefore control its execution order during replay. We also insert our own library code `Pre_RClassify.js`, to the HTML *head* element. This library code block will be executed before the browser loads the body of the web page, thereby allowing us to track the execution of all racing events.

Our instrumentation is designed to collect information about the race-condition warnings. Fig. 5 shows an example, where Lines 3 and 10 are added during our instrumentation. They execute `Pre_RClassify.js` at the start of the *head* element, to redefine API functions such as *addEventListener()* and *setTimeOut()* so we can intercept these function calls at run time. Specifically, we replace them with wrapper functions, which invoke the original APIs but also gather information of the racing events for later phases of our algorithm.

We also insert a *window.onload* event handler at the end of the HTML *body* element, to be fired after the page is fully loaded. Our function *RC_fire_handlers()* goes through all registered event handlers that require user actions, and fires them one by one, to simulate user interactions. By automatically dispatching these event handlers, as opposed to relying on clicks from the user [21], we have improved the coverage compared to existing race detectors.

B. Analyzing Race-condition Warnings

Another input of RCLASSIFY is the set of race-condition warnings. In our work, the race detector is EVENTRACER, which uses a modified version of WEBKIT to generate a trace log while loading the web page. Then, it analyzes the trace log to build a happens-before model to capture global resources that have conflicting accesses. Two accesses are *conflicting* if they refer to the same memory location and at least one of them is a write operation. However, the happens-before model in EVENTRACER is not precise enough to separate harmful races from bogus/harmless ones [22]. Our work focuses on precisely classifying these race-condition warnings.

Toward this end, we statically analyze the warnings to compute the information required by deterministic replay. We use JSOUP to create two instrumented versions of the web application. In one version, the self-control capability ensures $ev_a \prec ev_b$; in the other version, it ensures $ev_b \prec ev_a$. To prevent interference from other pairs of racing events, we force all other racing events to maintain their original execution order instead of allowing them to interleave freely.

```
1  <html>
2   <head>
3     <!-- load replay info of all races here ... -->
4     <script src="RClassify.js"></script>
5     ...
6    </head>
7    <body>
8     ...
9     <!-- end of the element m -->
10    <script> RC_detect_handler_changes(); </script>
11    ...
12    <!-- end of all body elements -->
13    <script> window.addEventListener("load",
            RC_dump_state, false); </script>
14   </body>
15 </html>
```

Fig. 6. Example: Instrumented web page prior to deterministic replay.

For example, given two warnings $RC_1(ev_a, ev_b)$ and $RC_2(ev_c, ev_a)$, the first execution may be $\lambda_1 = ps_0 \dots ps_i \xrightarrow{ev_c} \dots ps_j \xrightarrow{ev_a} \dots ps_k \xrightarrow{ev_b} \dots ps_n$ where $ev_a \prec ev_b$. To classify RC_1, we obtain the second execution $\lambda_2 = ps_0 \dots ps_i \xrightarrow{ev_c} \dots ps_j \xrightarrow{ev_b} \dots ps'_k \xrightarrow{ev_a} \dots ps'_n$, where $ev_b \prec ev_a$. In both executions, we try to maintain the order $ev_c \prec ev_a$ for RC_2.

Fig. 6 shows an example of the instrumented HTML file. Similar to Fig. 5, it loads another of our libraries, `RClassify.js`, at the start of the *head* element, as well as information of the racing events. This library contains functions to control the execution order of racing events and record the program state. We also insert a JavaScript code snippet to invoke the function *RC_detect_handler_changes()* after the parsing of every HTML element (Line 10). It checks if the list of event handlers attached to any element of the DOM has changed. If new handlers have been added, we retrieve and instrument them so as to track their executions at run time.

When loading the racing events (Line 3), if ev_a needs to be executed before ev_b, we put ev_a into the $toWaitList[ev_b]$. During replay, we monitor all racing events dynamically and force ev_b to wait for all events in $toWaitList[ev_b]$ before executing ev_b. For multiple warnings, however, it is not always possible to maintain the execution order of all other racing events while flipping the race condition under investigation. This is because reversing the order of a race condition may invalidate the order of other race conditions. For example, consider $RC_1(ev_a, ev_c)$, $RC_2(ev_a, ev_b)$ and $RC_3(ev_b, ev_c)$ in an execution where $ev_a \prec ev_b \prec ev_c$. When reversing RC_1, maintaining the original order of RC_2 and RC_3 becomes impossible.

In such cases, we try to maintain the order of as many of the other races as possible. That is, after loading the ordering information of the reversed RC_1 and the original RC_2, we discover that they conflict with the original order of RC_3. Therefore, we ignore RC_3 and obtain an execution where $ev_c \prec ev_a \prec ev_b$.

When the must-happens-before relation among race conditions are available, we use it to further refine the execution order information needed for replay. For example, if there is a must-happens-before relation $ev_a \rightarrow_{mhb} ev_b$, we know the order of ev_a and ev_b cannot be flipped. Although analyzing RC_1 or RC_2 alone would not detect any conflict, when reversing RC_1, we know that it is no longer possible to maintain both RC_2 and RC_3. Therefore, we ignore RC_2 and obtain an execution where $ev_c \prec ev_a \prec ev_b$.

We also insert a *window.onload* handler at the end (Line 13). The function `RC_dump_state` saves all relevant fields of the global state into a disk file. After recording the two program states, we compare them to decide if the race is harmful. We say (ev_a, ev_b) is harmful if the two states are significantly different. Toward this end, an important problem is to identify fields that may be affected by sources of nondeterminism other than race conditions; failure to do so will lead to harmless races to be incorrectly classified as harmful races.

In the next sections, we explain in detail how to accurately control the order of racing events during replay and how to compare relevant fields of the program states.

V. Controlled Executions

To replay the racing events, we need to intercept them and control their execution order at run time.

A. Intercepting the Callback Functions

We want to intercept the registration, invocation, and removal of callback functions for all global events, timers, and AJAX requests. We replace each callback function with a wrapper that, prior to invoking the original function, checks if the invocation should be postponed.

Broadly speaking, there are two types of callback functions for handling events. Type 1 handlers are installed by setting a DOM element's attribute such as *el.onload* and *el.onclick*. Type 2 handlers, in contrast, are added and removed by calling *addEventListener()* and *removeEventListener()*, respectively. Each element may have multiple Type 2 handlers, stored in the browser as opposed to the DOM itself; as such, these handlers cannot be accessed by traversing the DOM tree. Third-party libraries such as JQuery may define their own APIs, such as *jQuery.bind()* and *jQuery.detach()*, for managing event handlers. But internally they still rely on the two aforementioned mechanisms.

The addition and removal of Type 1 event handlers (via attributes such as *el.onclick*) are difficult to intercept at run time, since it is possible for the parsing of any HTML element (e.g., script elements) to add or remove event handlers. In RClassify, we do not modify the underlying web browser or intercept the execution of each individual JavaScript instruction. Instead, our instrumentation is performed at the HTML file level. To solve this problem, we developed a unified framework for detecting changes to event handlers, which periodically scans the DOM and compares it with a copy of the DOM recorded during the previous scan. If there is any change in the DOM element's Type 1 event handler, e.g., the addition or modification of *el.onclick*, we will detect it.

In Fig. 6, for example, the scan is implemented using JavaScript in `RC_detect_handler_changes()`. By statically instrumenting the HTML file, at run time we can invoke this function right after the parsing of each HTML element or the execution of each callback function that may modify the DOM. To reduce the runtime overhead, we statically analyze the HTML elements to invoke this function only if needed, e.g., after the parsing of JavaScript code, embedded HTML, and other elements whose event handlers contain racing events.

Algorithm 1 Scanning and instrumenting event handlers.

```
RC_detect_handler_changes ( ) {
  for each ( $el \in$ document.all_elements ) {
    for each ( $eh\_type \in$ all_event_handler_types ) {
      if ( $el[eh\_type]$ has changed ) {
        $event\_str$ = compose_es ( $el.id$, $eh\_type$ );
        $orig\_func$ = $el[eh\_type]$;
        if ( is_racing_event ( $event\_str$ ) )
          $el[eh\_type]$ = Replace_callback ( $event\_str$, $orig\_func$ );
      }
    }
  }
}
```

Algorithm 2 Controlling execution of callback functions.

```
Replace_callback ( $event\_str$, $orig\_func$ ) {
  function $RC\_func$( ) {
    if ( $orig\_func$ is defined )
      if ( $\neg$ racing_event_finished_waiting ( $event\_str$ ) )
        $postponedEvent$.push ( $RC\_func$ );
      else
        $orig\_func$.call ( $arguments$ );
  }
  return $RC\_func$;
}
```

The pseudocode is shown in Algorithm 1, which first scans the DOM to identify any change of *el[eh_type]*, corresponding to the event *el.eh_type*, and then instruments the handler: *var orig_func = el.onclick; el.onclick =* `Replace_callback` *(event_str, orig_func)* { *generate and return RC_func; ...*}, where *RC_func* is a dynamically generated instance of *orig_func*. In addition to controlling the execution order, *RC_func* also invokes the original function *orig_func*. Internally, `Replace_callback()` creates the wrapper *RC_func* using *event_str* and *orig_func*, and uses it to replace the original function.

Type 2 event handlers are intercept by replacing *addEventListener()* and *removeEventListener()* with wrapper functions while loading *RClassify.js* in the HTML *head* element. Internally, the procedure executes *var* `orig_addEL` *= addEventListener; addEventListener = function* `new_addEL()` {`orig_addEL`*();...*} Therefore, at runtime, instead of invoking the original function, it would invoke `new_addEL()`. Instead of adding *orig_func* as an event handler, `new_addEl()` dynamically creates a wrapper function *RC_func* as the event handler.

B. Controlling the Execution Order

After a callback function is replaced with its wrapper function, the execution order can be controlled with respect to other racing events. Specifically, the callback is instrumented according to Algorithm 2, where *orig_func()* is replaced by a dynamically generated instance of *RC_func()*. We control the event execution order by conditionally postponing the corresponding callback functions. When a racing event is about to be dispatched, instead of executing the original function directly, we call the function *RC_func*, which in turn checks if the corresponding *orig_func* is ready to execute. Each event ev has a precomputed set of events ($toWaitList[ev]$) that need to be executed before ev. If an event needs to be postponed, the function *RC_func* will be put into the waiting list. The list is checked periodically to ensure that callbacks are executed immediately after they are ready.

To ensure that replay does not deadlock, we use a counter inside each *RC_func* to record the number of times it has been postponed (not shown in Algorithm 2 for brevity). When the counter reaches a certain threshold, say 200, we assume that it has waited too long and thus declare that replay failed. After that, it will stop waiting and enter a free run.

The reason why replay may fail is because some racing events simply cannot be flipped (e.g., bogus race). In this sense, our method can robustly handle bogus races. However, it is not the most efficient way to detect bogus races, since waiting for replay to fail consumes a significant amount of time. Therefore, we also developed a cheaper mechanism for identifying certain bogus races common in web applications. Recall that bogus races are largely due to limitations of race detectors in modeling happen-before relationships. Certain event handlers, such as *document.onDOMContentLoaded* and *window.onload*, have somewhat fixed execution time: they are fired either after the entire HTML is parsed, or after all resources are loaded. Therefore, if a reported race is between the parsing of an HTML element and *document.onDOMContentLoaded* or *window.onload*, we know for sure that it is a bogus race. In such case, we can skip replay because replay is guaranteed to fail.

VI. Program State Comparison

We record the program states after both controlled executions and then compare their fields.

A. State Recording

After the web page is fully loaded, we serialize the program state and store the result in a disk file. We initiate state recording when the following two conditions are met: (1) the web page is fully loaded, and (2) all racing events have finished executing. We consider the following fields as parts of the program state:

1) *The DOM*: We record the value of all HTML elements, their attributes, and Type 1 event handlers. We also intercept all Type 2 event handlers and store them in a special DOM attribute.
2) *JavaScript variables*: We record the value of all JavaScript global variables since they affect the behavior of the application.
3) *Environment variables*: We record the value of environment variables associated with the browser, such as the height and width of the window.
4) *Console messages*: We record runtime information displayed in *console.log*, *console.warn* and *console.error*. They are invisible to end-users but useful to developers.

All JavaScript functions and variables used to implement RClassify are defined in a specific name space and thus can be excluded from the program state easily.

To serialize the data fields into a string, thus allowing them to be compared easily, we use the *JSON.stringify()* API. However, JSON does not directly handle data with cyclic dependency, which are common in web applications. To solve this problem, we implemented a JavaScript method that traverses the DOM object in the BFS order and marks each visited node with a unique identifier. If it encounters a visited object again, it replaces the reference to that object with its unique identifier. Since the resulting representation is guaranteed to be acyclic, it can be serialized to a string using *JSON.stringify()*.

B. State Comparison

To compare two program states, we first use *JSON.parse* to restore the data fields from disk files and construct two key-value tables. For the DOM, we use the element's *id* attribute as the key (the unique *id* generated by our HTML instrumentation) and the HTML content as the value. For JavaScript variables and environment variables, we use the variable name as the key. For console messages, we use their order in the console as the key. We say the race condition is *harmless* if the two states are identical. We say the race condition is *harmful* if they differ in the DOM or JavaScript global variables. If they differ only in console logs or the value of some environment variables, we assume the race condition is likely *harmless* but we still report it as a warning.

Sometimes, certain fields of the DOM or JavaScript variables may have nondeterministic values due to reasons other than race conditions. For example, an application may record the last time it is executed or create a session ID that is different every time it is executed. To exclude such fields in state comparison – since they may lead to false positives – we developed a mechanism for users to specify which fields should be excluded. We also developed a heuristic method for filtering out such irrelevant fields automatically.

Our solution is to execute the web application three times, with the following event orders: $ev_a \prec ev_b$, $ev_a \prec ev_b$, and $ev_b \prec ev_a$. After that, we use a three-way comparison to check the state differences. If there exists a field that has different values in all three states, then we consider it as an irrelevant field. If a field has the same value in the first two states, but a different value in the third state, then we consider it as a relevant field.

VII. Experiments

We have implemented our *evidence-based* classification method in a software tool (RClassify), which builds upon a number of open-source software. Our libraries for monitoring and controlling racing events is written in 2.1K lines of JavaScript code. Our front-end for HTML instrumentation is implemented using *Jsoup 1.8.1* in 2.4K lines of Java code. We leverage EventRacer to generate the race-condition warnings (our input). During the experiments, we store benchmarks under test in a local web server. We use *Teleport Ultra 1.69* to download the source files of real websites before instrumenting them. We use *Mozila Firefox* as the web browser – we experimented with versions ranging from 35.0 to 47.0 and did not encounter any incompatibility issue.

Our benchmarks fall into two groups. The first group consists of standard web application benchmarks from recent publications such as EventRacer [21], [22] and Wave [7]. Since they are written specifically for illustrating various types of race conditions, most of them have known-to-be-harmful races, and our goal is to confirm that RClassify can correctly identify them. The second group consists of seventy real websites randomly chosen from the portals of the Fortune-500 companies. Previously, EventRacer was applied to the same type of websites and reported many bogus warnings. Therefore, our goal is to see if RClassify can do better than state-of-the-art techniques, which include not only EventRacer [22], but also Mutlu et al. [18] and R4 [9].

That is, can RCLASSIFY accurately classify these warnings while maintaining a low runtime cost? Our experiments were conducted on a machine with Intel Core i7-4700 2.4 GHz CPU and 4 GB RAM running 64-bit Ubuntu.

A. *Results on Standard Benchmarks*

Table I shows our results on the standard benchmarks. Columns 1-2 show the benchmark name and number of race-condition warnings (input of RCLASSIFY). Column 3 shows the number of bogus races that RCLASSIFY identified. Columns 4-8 show the number of harmful races, harmless races, and undetermined cases (Undet), respectively. Column 9 shows the total time taken by RCLASSIFY. Note that some warnings cannot be generated by the race detector EVENTRACER due to its own limitations; for these warnings, we manually created them and gave them to RCLASSIFY. We checked all 50 input warnings manually and confirmed that RCLASSIFY produced the correct classification.

Specifically, the 13 harmless races fall into three groups: *none* means there are no differences in the program states; *con.* means there are differences in *console* logs only; and *b/c/t* means the differences are due to event "bubbling" and "capturing" or a "timer-try" pattern. Event bubbling and capturing (b/c) lead to duplicated events on the upper-level elements. For example, if a windows has 100 buttons, each of the 100 *button.onclick* events can be bubbled up to trigger *document.onclick*. In such case, there will be 100 races reported on *document.onclick*. Timer-try (t) is when a function repeatedly postpones its execution via *setTimeout()* until some condition is satisfied. The nature of these races means they are likely harmless.

We also downloaded and applied the tools from Mutlu et al. [18] and Jessen et al. [9] (R4). Mutlu et al. [18] was designed to report only harmful races, but it reported 0 races despite that 33 of the 50 races were known-to-be-harmful. R4 [9] focuses on applying *stateless model checking* to web applications to systematically explore the event interleavings, but also can filter race-condition warnings. Our experiments showed that

- R4 detected only 21 race conditions, among which it classified 8 as harmful (H), 1 as normal-risk (N), and 12 as low-risk (L).
- Our manual inspection showed that, whereas the 8 races marked as harmful (H) by R4 are indeed harmful, 3 of the other 13 races (which were not marked as harmful by R4) are also harmful.

Altogether, R4 reported only 8 (out of the 33) harmful races while missing the other 25.

B. *Results on Real Websites*

Table II shows the results of our experiments on real websites. Since going through all web portals of the Fortune-500 companies takes too much time, we randomly selected seventy companies and applied EVENTRACER to generate the initial set of race-condition warnings. Among them, twenty websites do not have any EVENTRACER-reported warnings; for the remaining fifty websites, the results are shown in the table. Columns 1-2 show the website name and the number of EVENTRACER-reported warnings (input of RCLASSIFY).

TABLE I
EXPERIMENTAL RESULTS ON THE STANDARD BENCHMARKS.

Name	Warning (input)	Bogus	Harmful	Harmless con.	Harmless b/c/t	Harmless none	Undet.	Time	From[18] Harmful	From[9] H/N/L
WebR_ex1	1	0	0	1	0	0	0	23.8s	0	0/1/0
WebR_ex2	1	0	1	0	0	0	0	23.8s	0	0/0/0
WebR_ex3	1	0	1	0	0	0	0	23.8s	0	0/0/1
WebR_ex4	1	0	1	0	0	0	0	23.8s	0	0/0/0
WebR_ex5	1	0	0	1	0	0	0	24.1s	0	0/0/0
EventR_ex1	1	0	0	1	0	0	0	23.8s	0	1/0/0
EventR_ex2	1	0	1	0	0	0	0	23.8s	0	0/0/1
EventR_tut	3	0	3	0	0	0	0	1m10s	0	0/0/0
kaist_ex1	1	0	0	1	0	0	0	23.8s	0	0/0/1
kaist_ex2	1	0	0	1	0	0	0	25.0s	0	1/0/0
kaist_ex3	3	0	1	2	0	0	0	1m10s	0	1/0/0
kaist_case1	6	0	4	1	1	0	0	2m20s	0	1/0/2
kaist_case2	3	0	3	0	0	0	0	47.2s	0	1/0/1
kaist_case3	4	1	2	1	0	0	0	1m35s	0	1/0/0
kaist_case4	7	4	3	0	0	0	0	1m51s	0	0/0/3
kaist_case5	3	0	3	0	0	0	0	1m11s	0	0/0/1
kaist_case6	3	0	2	1	0	0	0	24.3s	0	1/0/0
kaist_case7	7	0	6	1	0	0	0	2m50s	0	0/0/1
kaist_case8	2	0	2	0	0	0	0	47.6s	0	1/0/1
Total	**50**	5	**33**	11	1	0	0		**0**	**8**/1/12

```
<script id="script_1">
 function show() {
  document.getElementById("dw").style.display="block";
 }
</script>
<a id="href1" href="javascript:show()"> link </a>
<div id="dw" style="display:none"> dw </div>
```

Fig. 7. A harmful race condition RCLASSIFY identified but R4 [9] missed.

Columns 3-8 show the number of bogus races, harmful races, harmless races, and undecided cases, respectively. Column 9 shows the total time taken by RCLASSIFY). Finally, Columns 10-12 show the results reported by R4.

While using EVENTRACER to generate the input warnings, we found that some of the reported races do not make sense since the racing events have empty read/write accesses (likely due to defects in EVENTRACER); therefore, we filtered them out before applying RCLASSIFY–they are labeled *Undet.* in the table.

The results in Table II show that RCLASSIFY identified 132 harmful races out of 1,903 EVENTRACER reported warnings. We manually inspected the state comparison results of these races and confirmed the correctness of all classifications. An example of the harmful races identified by RCLASSIFY is already shown in Fig. 4, which turns out to be a real bug from www.newyorklife.com. In contrast, R4 identified only 33 harmful races. There are also six websites on which R4 crashed – they are marked as "(CRASH)" in the table.

A closer investigation shows that R4 relies on the severity of error logs and exception logs to determine the risk level of each race. Although R4 saves a picture screenshot after each execution for the user's reference, it does not record the program state. Even if the developers manually compare screenshots, it is not as accurate or informative as comparing the program states. For example, in Fig. 7, R4 failed to identify the harmful race (occurred when *href1* is clicked before the *dw* element is parsed, thus causing *show()* to access a non-existing element) but there is no visual difference. In contrast, RCLASSIFY can detect the difference in program states and thus identify it as a harmful race.

While reviewing the classification results, we also noticed that harmful race conditions from the same website were often

TABLE II
EXPERIMENTAL RESULTS ON REAL WEBSITES RANDOMLY CHOSEN FROM WEB PORTALS OF FORTUNE-500 COMPANIES.

Name	Warning (input of RCLASSIFY)	Bogus	Harmful	Harmless			Undet.	Time	Result of R4 [9]		
				console	b/c/t	none			Harmful	Normal	Low
www.aa.com	7	0	3	3	0	1	0	5m30s	2	7	24
www.adp.com	2	0	1	0	1	0	0	43.8s	8	14	15
www.ally.com	91	0	8	71	0	8	4	67m48s	0	1	1
www.altria.com	13	2	3	2	4	1	1	4m5s		(CRASH)	
www.amazon.com	10	6	0	0	3	0	1	0		(CRASH)	
www.arrow.com	1	1	0	0	0	0	0	0	1	2	3
www.ashland.com	6	0	0	0	1	5	0	3m17s		(CRASH)	
www.autozone.com	9	4	1	0	2	0	2	41.6s	3	4	20
www.ball.com	2	0	0	2	0	0	0	1m14s	0	0	1
www.bd.com	22	2	5	15	0	0	0	11m16s	1	1	1
www.biogen.com	26	1	0	6	0	4	15	7m40s		(CRASH)	
www.boeing.com	4	1	0	1	0	2	0	1m39s	0	1	4
www.buckeye.com	23	1	1	4	0	16	1	12m53s	0	3	35
www.campbellsoupcompany.com	5	3	1	1	0	0	0	1m16s	0	0	3
www.coach.com	95	1	13	61	0	18	2	75m18s	0	0	1
www.cognizant.com	56	2	25	26	2	0	1	73m37s	0	1	2
www.disney.com	1	0	0	1	0	0	0	1m8s	0	4	4
www.dollartree.com	501	0	6	14	238	243	0	235m31s	0	8	5
www.freddiemac.com	1	1	0	0	0	0	0	0	0	0	0
www.generalcable.com	7	1	1	4	0	1	0	3m32s	0	0	1
www.heinz.com	3	1	0	2	0	0	0	1m22s	0	0	1
www.honeywell.com	1	0	1	0	0	0	0	0m36s	2	6	8
www.huntsman.com	70	2	18	0	0	45	5	21m18s	0	0	0
www.l-3com.com	404	0	0	0	109	294	1	161m18s	0	3	7
www.loews.com	4	3	1	0	0	0	0	37.6s	1	4	0
www.marathonoil.com	1	0	0	0	0	1	0	30.3s	0	2	2
www.mcdonalds.com	4	1	2	1	0	0	0	5m58s	2	0	5
www.newyorklife.com	18	13	2	1	1	0	1	1m49s	2	2	16
www.nov.com	5	1	0	1	3	0	0	1m1s	0	0	32
www.pge.com	281	0	0	13	99	169	0	126m30s	0	0	2
www.regions.com	14	8	0	4	0	1	1	3m33s	0	9	2
www.sandisk.com	5	2	1	1	0	1	0	1m42s	2	1	6
www.simon.com	1	1	0	0	0	0	0	0	0	1	0
www.sjm.com	12	0	1	5	3	3	0	4m59s	1	0	2
www.starbucks.com	8	2	1	0	4	1	0	1m38s	0	1	4
www.target.com	9	2	0	0	3	0	4	0	1	2	6
www.tractorsupply.com	7	1	0	1	0	0	5	43.2s	1	1	3
www.trw.com	8	0	0	0	3	5	0	2m48s	0	4	3
www.tysonfoods.com	1	0	0	1	0	0	0	53.7s	0	0	6
www.unfi.com	13	0	3	1	0	1	8	3m22s		(CRASH)	
www.unitedhealthgroup.com	11	1	0	7	3	0	0	4m10s	3	0	2
www.unitedrentals.com	7	1	1	4	0	0	1	3m46s	1	0	7
www.unum.com	1	0	0	1	0	0	0	31.5s		(CRASH)	
www.valero.com	36	2	8	1	23	2	0	6m41s	1	5	1
www.wd.com	2	1	0	0	1	0	0	0	1	0	1
www.wdc.com	2	1	0	0	1	0	0	0	0	1	1
www.wesco.com	5	1	3	1	0	0	0	2m3s	0	0	1
www.weyerhaeuser.com	29	0	21	8	0	0	0	14m45s	0	0	0
www.wfscorp.com	16	1	0	8	0	6	1	7m33s	0	2	1
www.yum.com	43	2	1	1	11	28	0	15m36s	0	0	1
Total	**1,903**	73	**132**	273	515	856	54		**33**	90	240

correlated: fixing one race (e.g., by adding a must-happen-before constraint) also fixes many of the other races.

The execution time of RCLASSIFY is largely determined by how fast the browser loads the target web page, since we need to wait for the page to be completely loaded before recording the program state. In practice, it is common for a website to have large resource files that take a significant amount of time to download. An example is www.coach.com, which may take tens of seconds to load completely and thus increase the total execution time of RCLASSIFY. In addition, we inserted several sleep commands in our framework to ensure a smooth connection between operations, e.g., between the script for saving the state recording file and the script for loading the next web page. During state comparison, we also run the same web application three times, among which two are in the original execution order while the third one is with the racing events flipped. For each individual replay, we observed only a slowdown of 1.5-2X when compared to a free run. Also note that we have not optimized the runtime performance of our tool. Nevertheless, RCLASSIFY is fully automated, and therefore is significantly more efficient than manually classifying race-condition warnings.

C. Compared to Heuristic Filtering

We also compared RCLASSIFY with EVENTRACER's heuristic filtering, which does not check the effect of racing events, but instead relies on event types and a limited number of bug patterns. As such, they may report false positives as well as miss harmful races. More specifically, EVENTRACER has two levels of filtering. The first-level filtering aims at removing *benign* races. The second-level filtering aims at identifying *high-risk* (harmful) races. We use *the-rest* to denote left-over races after these two levels of filtering.

Table III shows our experimental results. Of the 1,903 race-condition warnings, EVENTRACER's heuristic filtering identified 558 as high-risk (harmful), 372 as benign, and 973 as the-rest. In contrast, our method showed that only 38 of the 558 high-risk (harmful) races are truly harmful, whereas 40 of the 372 benign races are harmful. In addition, 54 of the 973 left-over races are also harmful. The results show that heuristic filtering techniques are not effective in practice.

Fig. 8 shows a harmful race condition detected by RCLASSIFY from the official website of Honeywell International, but considered by *EventRacer*'s heuristic filtering as *benign* (they call it *filtered*). The race condition is between the parsing of

TABLE III
RCLASSIFY VERSUS EVENTRACER'S HEURISTIC FILTERING [22].

Total	EVENTRACER's *high-risk*	EVENTRACER's *benign*	EVENTRACER's *the-rest*
1,903	558	372	973
Harmful	**38** (6.8%)	**40** (10%)	**54** (5.5%)

asynchronous script *jsapi* (ev_a) and the parsing of synchronous script *gssAutoComplete.js* (ev_b). By executing these racing events in different orders, RCLASSIFY detected differences in many fields of the program states. Among them, one difference is in the title field of the web page: When $ev_b \prec ev_a$, the title is *"Honeywell - Global Technology Leader in Energy Efficiency, Clean Energy Generation, Safety & Security, and Globalization"*, but when $ev_a \prec ev_b$, the title is *"Home"*.

```
<!-- other elements -->
<script id="script_4">
  ...
  var gjsapi = document.createElement('script');
  gjsapi.type = 'text/javascript';
  gjsapi.src = ('https:' == document.location.protocol
        ? 'https://' : 'http://') +
        'www.google.com/jsapi';
  var s = document.getElementsByTagName('script')[0];
  s.parentNode.insertBefore(gjsapi, s);
  ...
</script>
<!-- other elements -->
<script type="text/javascript" id="script_26"
      src="http://honeywell.com/_layouts/
InternetFramework/Scripts/gssAutoComplete.js"></script>
<!-- other elements -->
```

Fig. 8. A harmful race condition filtered out by EVENTRACER [22].

VIII. RELATED WORK

There are several existing tools for detecting race conditions in web applications, some of which are based on conservative static analysis (e.g., Zheng et al. [33] and ARROW [32]) while others are based on dynamic analysis [22], [21], [7]. Static analysis has the advantage of covering all possible execution paths of the JavaScript code, but due to the rich set of dynamic features in JavaScript and client-side web applications in general [6], [10], [23], [28], [16], [29], they cannot robustly handle real websites. Dynamic analysis tools such as EVENTRACER [22], [21] and WAVE [7] do not have such limitations. However, they have limited code coverage and often report many bogus warnings as well as miss real bugs. Furthermore, none of these existing methods uses *evidence-based* techniques to classify race conditions.

Mutlu et al. [18] proposed a method for detecting race conditions using a combination of dynamic and static analysis techniques. They obtain an execution trace and then use predictive static analysis to detect races that may occur in all possible interleavings of events of the given trace. However, their method does not use deterministic replay to check the actual impact of racing events and therefore may still report bogus and harmless races. Furthermore, their tool is limited to detecting certain types of races over persistent storage and may miss many other races, as shown in our experiments.

Jensen et al. [9] proposed a stateless model checking tool (R4) for systematically exploring event interleavings in a web application. Unlike tools such as EVENTRACER [22], [21], which detect races only in one trace, R4 can generate many new traces from the given trace and detect races in these traces. However, as shown in our experimental evaluation, R4 does not perform well in classifying the races: it missed many harmful races and reported many false positives.

There are also software tools for recording the state of a running web application [17], [2], [5], but the goal is to allow developers to revisit the recorded program state to diagnose bugs, or dynamically migrate the web application to other platforms [4], [15], [14]. Therefore, although they also address the state recording problem, the applications are significantly different from ours: none of these prior works focuses on diagnosing concurrency bugs.

For diagnosing *data-races* in multithreaded programs, there is a large body of work [19], [25], [31], [26], [11], [24], [3], [13], [12]. Specifically, Narayanasamy et al. [19] proposed perhaps the first deterministic replay-based method for classifying data-races in multithreaded applications. They used a checkpointing tool to take snapshots of the main memory while executing the program, and compared the snapshots to decide if a data-race is harmful. Similar works also include Sen's race-directed testing tool [25] and the *Portend* tool by Kasikci et al. [11]. There are also many testing tools such as CTrigger [20], PENELOPE [27], Fusion [30], and RV-Predict [8] for multithreaded programs.

However, *data-races* in multithreaded C/C++ or Java programs are significantly different from *race conditions* in client-side web applications. Specifically, the computing platforms are significantly different in that one relies on *multi-threading* whereas the other relies on *event-driven* execution. The supporting tools are also different. For multithreaded programs, there already exist a large number of checkpointing tools, but for web applications, we are not aware of any such checkpointing tool. Due to these reasons, both deterministic replay and program-state comparison require drastically different solutions. Therefore, except for the high-level similarity, RCLASSIFY is completely different from these existing methods and tools.

IX. CONCLUSIONS

We have presented RCLASSIFY, the first *evidence-based* method for automatically classifying race-condition warnings in web applications. It identifies the real and harmful races based on executing the racing events in different orders and then comparing the program states. We also developed a purely JavaScript-based, platform-agnostic framework for monitoring and controlling the execution order of racing events. We have implemented RCLASSIFY and evaluated it on both standard benchmarks and a large set of real websites. Our experimental results show that the new method is both effective in identifying harmful races and scalable for practical use. For future work, we plan to leverage RCLASSIFY not only to diagnose race conditions but to automatically repair them.

X. ACKNOWLEDGMENTS

This work was primarily supported by the National Science Foundation (NSF) under the grants CCF-1149454, CCF-1405697, and CCF-1615834. Partial support was provided by the Office of Naval Research (ONR) under the grant N00014-13-1-0527.

References

[1] Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. Understanding JavaScript event-based interactions. In *International Conference on Software Engineering*, pages 367–377, 2014.

[2] Silviu Andrica and George Candea. WaRR: A tool for high-fidelity web application record and replay. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 403–410, 2011.

[3] Mitra Tabaei Befrouei, Chao Wang, and Georg Weissenbacher. Abstraction and mining of traces to explain concurrency bugs. In *International Conference on Runtime Verification*, pages 162–177, 2014.

[4] Federico Bellucci, Giuseppe Ghiani, Fabio Paternò, and Carmen Santoro. Engineering JavaScript state persistence of web applications migrating across multiple devices. In *ACM SIGCHI Symposium on Engineering Interactive Computing System*, pages 105–110, 2011.

[5] Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. Interactive record/replay for web application debugging. In *ACM Symposium on User Interface Software and Technology*, pages 473–484, 2013.

[6] Salvatore Guarnieri and V. Benjamin Livshits. GATEKEEPER: mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium*, pages 151–168, 2009.

[7] Shin Hong, Yongbae Park, and Moonzoo Kim. Detecting concurrency errors in client-side java script web applications. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 61–70, 2014.

[8] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 337–348, 2014.

[9] Casper Svenning Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin T. Vechev. Stateless model checking of event-driven applications. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 57–73, 2015.

[10] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the Eval that men do. In *International Symposium on Software Testing and Analysis*, pages 34–44, 2012.

[11] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: Telling the difference with portend. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 185–198, 2012.

[12] Sepideh Khoshnood, Markus Kusano, and Chao Wang. ConcBugAssist: Constraint solving for diagnosis and repair of concurrency bugs. In *International Symposium on Software Testing and Analysis*, 2015.

[13] Markus Kusano, Arijit Chattopadhyay, and Chao Wang. Dynamic invariant generation for concurrent programs. In *International Conference on Software Engineering*, 2015.

[14] James Teng Kin Lo, Eric Wohlstadter, and Ali Mesbah. Imagen: Runtime migration of browser sessions for javascript web applications. In *International World Wide Web Conference*, pages 815–826, 2013.

[15] James Teng Kin Lo, Eric Wohlstadter, and Ali Mesbah. Live migration of JavaScript web apps. In *International World Wide Web Conference*, pages 241–244, 2013.

[16] Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 499–509, 2013.

[17] James W. Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for JavaScript applications. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 159–174, 2010.

[18] Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. Detecting JavaScript races that matter. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 381–392, 2015.

[19] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 22–31, 2007.

[20] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: Exposing atomicity violation bugs from their hiding places. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36, 2009.

[21] Boris Petrov, Martin T. Vechev, Manu Sridharan, and Julian Dolby. Race detection for web applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 251–262, 2012.

[22] Veselin Raychev, Martin T. Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 151–166, 2013.

[23] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do - A large-scale study of the use of eval in JavaScript applications. In *European Conference on Object-Oriented Programming*, pages 52–78, 2011.

[24] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. Generating data race witnesses by an SMT-based analysis. In *NASA Formal Methods*, pages 313–327, 2011.

[25] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 11–21, 2008.

[26] Arnab Sinha, Sharad Malik, Chao Wang, and Aarti Gupta. Predictive analysis for detecting serializability violations through trace segmentation. In *International Conference on Formal Methods and Models for Co-Design*, pages 99–108, 2011.

[27] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. PENELOPE: Weaving threads to expose atomicity violations. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 37–46, 2010.

[28] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *European Conference on Object-Oriented Programming*, pages 435–458, 2012.

[29] Chungha Sung, Markus Kusano, Nishant Sinha, and Chao Wang. Static DOM event dependency analysis for testing web applications. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2016.

[30] Chao Wang, Mahmoud Said, and Aarti Gupta. Coverage guided systematic concurrency testing. In *International Conference on Software Engineering*, pages 221–230, 2011.

[31] Chao Wang, Yu Yang, Aarti Gupta, and Ganesh Gopalakrishnan. Dynamic model checking with property driven pruning to detect race conditions. In *International Symposium on Automated Technology for Verification and Analysis*, pages 126–140, 2008.

[32] Weihang Wang, Yunhui Zheng, Peng Liu, Lei Xu, Xiangyu Zhang, and Patrick Eugster. ARROW: Automated repair of races on client-side web pages. In *International Symposium on Software Testing and Analysis*, 2016.

[33] Yunhui Zheng, Tao Bao, and Xiangyu Zhang. Statically locating web application bugs caused by asynchronous calls. In *International Conference on World Wide Web*, pages 805–814, 2011.