Detecting JavaScript Races That Matter

Erdal Mutlu
Koç University, Turkey

Serdar Tasiran
Koç University, Turkey

Benjamin Livshits
Microsoft Research, USA

ABSTRACT

As JavaScript has become virtually omnipresent as the language for programming large and complex web applications in the last several years, we have seen an increase in interest in finding data races in client-side JavaScript. While JavaScript execution is single-threaded, there is still enough potential for data races, created largely by the non-determinism of the scheduler. Recently, several academic efforts have explored both static and runtime analysis approaches in an effort to find data races. However, despite this, we have not seen these analysis techniques deployed in practice and we have only seen scarce evidence that developers find and fix bugs related to data races in JavaScript.

In this paper we argue for a different formulation of what it means to have a data race in a JavaScript application and distinguish between benign and harmful races, affecting *persistent* browser or server state. We further argue that while benign races — the subject of the majority of prior work — do exist, harmful races are exceedingly rare in practice (19 harmful vs. 621 benign). Our results shed a new light on the issues of data race prevalence and importance.

To find races, we also propose a novel lightweight runtime symbolic exploration algorithm for finding races in traces of runtime execution. Our algorithm eschews schedule exploration in favor of smaller runtime overheads and thus can be used by beta testers or in crowd-sourced testing. In our experiments on 26 sites, we demonstrate that benign races are considerably more common than harmful ones.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*monitors, symbolic execution*

General Terms

Reliability

Keywords

JavaScript, asynchrony, race detection, non-determinism

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786820

1. INTRODUCTION

JavaScript is used widely in client-side Web programming. JavaScript execution is single-threaded. Yet the complex needs of sites such as Facebook, Outlook, and Google Maps have led to asynchrony becoming a common way to program complex Web applications. It is asynchronous processing that has made possible responsive user interfaces (UIs), different from the web applications of the late 1990s that required reloads. Despite JavaScript lacking conventional threads, the presence of asynchrony creates a potential for races. In particular, the ordering of event execution in JavaScript as well as the timing of completions of asynchronous requests is non-deterministic, affected by, e.g., network delays resulting in data races.

In this paper we argue that one should distinguish between races that have persistent consequences and those that are ephemeral. We find that the majority of races in JavaScript have no persistent consequences — we dub these races *benign.* This is because these races only result in invisible to the user portions of the program state, or, at worst, UI glitches that are either unnoticed by the user, or disappear if the user reloads the page. Given the forgiving nature of JavaScript execution, where failures of individual event handlers force the scheduler to terminate the current event handler and move execution to the next one, these kinds of failures are not as important as previously believed. We suggest that a more useful way to think about data races on the web is by focusing on *persistent state*, such as client-local cookies, `localStorage` and `sessionStorage` mechanisms, as well as server-based side efforts. The latter are achieved via `POST` calls to the server (`GET` calls are designed for reads and are supposed to be idempotent, and thus are not frequently used for state updates). Our experiments confirm that the number of such *harmful* races is quite modest, yet it is these kinds of races that are more likely to be considered serious and fixed by developers. Previous research efforts tend to produce a large number of reports, despite the emphasis on suppressing false positives [13, 11, 17].

It is not our goal to provide a sound over-approximation of all possible races in a given application or site, but, instead to provide a *lightweight exploration* algorithm that allows for the exploration of multiple schedules while only requiring a single run. This approach can be used for testing, including collaborative testing by a large number of beta- or crowd-sourced testers.

Contributions: We make the following contributions:

- We propose a new view of benign and harmful data races in JavaScript web applications, and argue that harmful races should be the primary focus of analysis tools, due to them affecting the persistent client- or server-side state of the applications.

- We propose a lightweight exploration algorithm for finding data races in runtime traces of JavaScript programs. A key advantage for the scalability of our approach is that it does not require multiple program runs and can operate on the basis of a single execution.
- We find and investigate a total of 19 harmful and 621 benign races in 26 web sites, with only 2 observed false positives.

2. OVERVIEW

There is increased interest in data races in asynchronous programs and, in particular, JavaScript. Static and runtime methods have been explored. A fundamental challenge with static analysis for JavaScript is that it is quite difficult to even *enumerate* all the relevant code, as much of the JavaScript code is produced with the help of `eval` calls and dynamic code loading. As a result, the traditional advantage of static analysis, namely, full path coverage largely does not apply to this problem. As such, the ability to make sound statements about the lack of data races is compromised [17, 3, 15, 6].

Runtime techniques in this space are also vulnerable to losing precision, which leads tool authors to develop heuristics to eliminate potential false positives [13, 9, 5]. They also suffer from the lack of *coverage* and the inability of making sound guarantees about the lack of races. Additionally, runtime techniques that involve combinatorial *schedule exploration* can run into scalability challenges, especially when the number of possible handlers to schedule is high [13, 9].

Our technique attempts to combine the advantages of static and runtime analysis. We execute the code only *once*, yet we explore *multiple* execution orders. As such, our technique scales well, while increasing the coverage of a single-pass runtime analysis. One way to see our approach is that it explores *neighboring* schedules for a particular runtime execution by analyzing possible re-orderings of the asynchronous event handlers. We foresee this approach as being especially useful in the context of beta- or crowd-sourced testing: having a large number of users will naturally increase code coverage. At the same time, the users' sessions will not be significantly slowed down. It should be noted that in the browser, slowing down the browser runtime runs the risk of modifying the behavior of timeout set with `setTimeout` and `setInterval`; additionally, the runtime may actively attempt to terminate slow-running events.

2.1 What is a Data Race?

Several possibly definitions of data races have been proposed for web applications [10, 11, 13]. They all center around the idea of writes to shared state that are performed by callbacks. Some of the races in prior work are caused by user interactions and browser-induced timing. In this work, our chief focus is on the `XmlHttpRequest` (XHR) mechanism, which allows client-side code to request data from servers:

```
var xhr = new XmlHttpRequest();
xhr.open("GET", "http://www.data.com/mydata.json");
xhr.onreadystatechange = function(e, d){
...
};
xhr.send(null);
```

The code above is for a typical `GET` request that obtains JSON data from a server and schedules an asynchronous `onreadystatechange` callback to process the data once it arrives. Multiple such callbacks can be outstanding, creating the possibility of what we dub an XHR-XHR race, if these callbacks write to shared state. Moreover, commonly, while synchronous XHR execution is possible, XHRs are scheduled to be dispatched asynchronously, to maintain a responsive client-side UI [10]. In the rest of the paper, we shall focus on asynchronous XHRs.

Secondly, a single XHR callback can race with the browser, resulting in the `state` variable being set to either 1 or 2. This is because the browser may have multiple script blocks, some of which may be scheduled either before or after the callback, depending on the callback's arrival and how fast the browser is rendering content:

```
<script>
var xhr = new XmlHttpRequest();
xhr.open("GET", "http://www.data.com/mydata.json");
xhr.onreadystatechange = function(e, d){
    state = 1;
};
xhr.send(null);
</script>
...a lot of text and images here...
<script>
    state = 2;
</script>
```

Thirdly, and even more subtly, if the user opens the same site in multiple browser *tabs*, it is possible for these tabs to lead to concurrent execution. Two instance of the code below may race with each other when run in different tabs, resulting in a cookie-based race on line 5:

```
1 <script>
2 var xhr = new XmlHttpRequest();
3 xhr.open("GET", "http://www.data.com/mydata.json");
4 xhr.onreadystatechange = function(e, d){
5     document.cookie = "value=" + Math.Random();
6 };
7 xhr.send(null);
8 </script>
```

The happens-before relation for asynchronous callbacks is defined by the creation order. The XHR callback is preceded by the code that creates the XHR (`xhr.send`). A specific case of this is what we call nested (or chained) XHRs, when callbacks are defined one within another. Practically, this is about the only way for the developer to ensure that there is ordering of XHR callbacks, so we see this programming pattern quite a bit.

2.2 Motivating Examples

In an effort to understand the possible impact of data races on the web, we spent some time analyzing bug reports for open-source projects located on GitHub. Below we describe some of the examples of subtle server-side bugs from GitHub. In the interest of fairness we should mention that these examples of races reported as GitHub issues were not particularly common bugs for JavaScript projects, an intuition that is largely confirmed by our results in Section 4.

Example 1 [Old server state.] Issue #79 for the WheatonWHALE project[1] describes the following situation:

1. The user reloads the page;

[1] `https://github.com/WheatonWHALE/whaleweb/issues/79`

```
<html>
 <script>                                    // 1
  var xhr = new Xhr();
  xhr.open('', false);
  xhr.onreadystatechange = function(){     // 2
    document.cookie = 'var1=1';
  };
  xhr.send();
  //for(i=0;i<10000000; i++) console.trace(i);
</script>
...
<!-- <input id='mydiv' /> -->
<script>                                     // 3
  document.cookie = 'var1=2';
</script>
...
<script>                                     // 4
  var xhr2 = new Xhr();
  xhr2.open('', false);
  xhr2.onreadystatechange = function(){    // 5
    document.cookie = 'var1=3';
  };
  xhr2.send();
</script>
</html>
Put your code here.
```

Figure 1: Multiple XHR example.

2. `onbeforeunload` listener fires, and the data is saved to server;
3. the page is loaded up again, and asks the server for the data;
4. the client-side JavaScript code loads up the old (outdated) data;
5. client state is saved to the server.

In the last step, the old, outdated data is saved to the server, essentially ignoring data updates. The culprit is the fact that steps 1 and 3 can race with each other: the data load request may arrive before the save is processed. The implemented fix makes data updates synchronous. □

Example 2 [Racing for a user ID.] A somewhat similar situation that has to do with the issue of stale data obtained from the server is captured in issue #20 in a project called LikeLines[2]. LikeLines provides users with an in-browser video player with a navigable heat map of interesting regions for the videos they are watching. This case describes two racing XHR calls that are issued to the back-end server during initialization by the following functions: 1) `createSession` and 2) `aggregate`. The first call is to create a new session for recording user interactions. The second call is needed for drawing a heat map.

The problem arises when a user has not contacted the backend server before. In this case, both XHR calls will be issued *without* a cookie, and in both cases the server will create a *new* user ID. This is clearly a problem because interaction sessions are tied to a user ID in this applications. If the cookie from the call to `aggregate` "wins" (i.e., arrives last), then subsequent calls to the server will contain a user ID that does not match the interactions session. □

In addition to the GitHub issues discussed above, below we list an illustrative example inspired by some of the samples from prior work [13, 17], although prior work did not focus on the issue of asynchronous XHRs.

Example 3 [Racing with the browser.] Consider the code in Figure 1. For convenience, we mark every handler above with a number. The happens-before relation induced by this code example is as follows: ① ← ②, ① ← ③, ③ ← ④, ④ ← ⑤. As such, our exploration algorithm will consider the possibility ① ← ② ← ③ and ① ← ③ ← ②. Similarly, because ② and ⑤ are weakly ordered, traces in which ② happens before or after ⑤ will be considered.

While one can explore these traces via a search in the schedule space, we choose to do so via data flow. We keep track of the event handlers that may be "concurrent" and mark the writes that they make to the same locations as weak writes.

Because ② and ③ can race, the value of `document.cookie` will be either `var1` = 1 or `var1` = 2. Similarly, for 2 and 5, the value of `document.cookie` will be either `var1` = 1 or `var1` = 5. To preserve precision, our algorithm maintains existing happens-before relations such as those between ① and ② and ① and ④. □

[2]`https://github.com/ShinNoNoir/likelines-player/issues/20`.

```
 1 READ PROP ID[99044816] = "open" :  JSFunction
 2 XHR [0000001] Open GET
 3 READ PROP ID[98817648] = "send" :  JSFunction
 4 XHR [0000001] Send
 5 XHR [0000001] Callback
 6 BEGIN XHR_Callback
 7  READ ID[235205312] = document :  JSObject
 8  WRITE PROP ID[242159440] = cookie :  JSInteger (1)
 9  Cookie [1020D800] Write "var1=1"
10 END XHR_Callback
11
12 READ ID[235205312] = document :  JSObject
13 WRITE PROP ID[242159440] = cookie : JSInteger (2)
14 Cookie [1020D800] Write "var1=2"
15
16 READ PROP ID[99044816] = "open" : JSFunction
17 XHR [0000002] Open GET
18 READ PROP ID[98817648] = "send" : JSFunction
19 XHR [0000002] Send
20 XHR [0000002] Callback
21 BEGIN XHR_Callback
22  READ PROP ID[108330176] = "readyState": JSInteger(4)
23  READ ID[235205312] = "document" :  JSObject
24  WRITE PROP ID[242159440] = "cookie" : JSInteger (3)
25  Cookie [1020D800] Write "var1=3"
26 END XHR_Callback
```

Figure 2: Sample trace illustrating cookie races.

2.3 Trace Processing

Figure 2 shows a simple trace obtained by running the code in Figure 1 that illustrates cookie-based races. We start processing the lines 1–4 where the XHR is opened and send to the remote server, we mark the XHR callback as an active callback which can be executed asynchronously any time in the future. As we process the first XHR callback on lines 6–10, we will record the write value made to cookie variable into the memory map where we store values for each variable id (i.e. `242159440` for cookie).

While recording the written value, we will look for values of the variable that are written by any callbacks that may be racing with each other, and complain about a race if there is any. As we continue processing the trace, we will record the value written on lines 12-14 by first checking the earlier values of `document.cookie`. As this sequential code segment can race with the earlier XHR callback (there is no happens-before edge), our processing will record a race on `document.cookie`, while adding a new value for the cookie into the memory map. As we continue to process the trace, a new XHR is opened and send to the server on lines 16–19 and added to the active callbacks list.

Later, the callback for the second XHR will be processed, when a write to the cookie is performed on lines 21–

26. While processing the write operation, the values for `document.cookie` will be checked and a race will be recorded, as two XHR callbacks are marked as racing, resulting in different cookie values.

2.4 Algorithm Summary

Here we provide the underlying intuition for our approach, with a more formal treatment relegated to Section 3. The key idea behind our approach is to consider alternative scenarios within a given trace. We do not attempt to force exploration of UI interactions, for example, however, we do explore the possibilities of different schedules that may occur because of the order of arrival of asynchronous event handlers that are part of the trace.

Our approach effectively performs static analysis on a trace that is collected at runtime, as a way to consider different schedules. When considering multiple execution of XHR callbacks as shown in Figure 3, the key observation is that instead of *separately* considering each of the possible schedules, we can *encode* the effect of the possible race by merging the state and keeping track of multiple, merged values. This is analogous to doing meets in static dataflow or abstract interpretation-style analysis, as an alternative to a costly meet-over-all-paths (MOP) solution. Conceptually, given a merge point for variable z with multiple values coming in for two racing XHRs, $z = v_1$ and $z = v_2$, we keep track of both values $\{v_1, v_2\}$; not of course that if $v_1 = v_2$, no need to keep two copies of the same values exists.

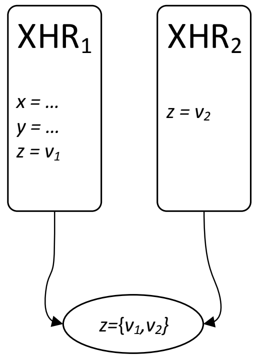

Figure 3: Merging two states after an XHR.

We formulate our race detection algorithm as a *dataflow analysis* on the values within a given execution trace. We flag a possible race if multiple values may flow to a sensitive location, indicating a presence of scheduling dependencies; these sensitive locations are persistent storage such as `document.cookie`, `localStorage`, `sessionStorage`, and, lastly, the DOM, etc. These latter locations serve as the *sinks* of our data flow analysis. Returning to the example in Figure 1, we can represent the race between handlers ② and ⑤ as an assignment `document.cookie` = {`'var1 = 2'`, `'var1 = 3'`} as a merge node after handler ⑤. This of course represents a direct flow of multiple values to a sensitive persistent storage location `document.cookie`. More interesting cases involve multiple steps of propagation.

2.5 Implementation

To collect execution traces, we have instrumented the most recent version of the Firefox web browser. Our changes span both the SpiderMonkey JavaScript engine to track data propagation through the memory of the browser as well as operations on `document.cookie`, `localStorage`, and the like, which are recorded by instrumenting the DOM. Our instrumentation spans over three main components of Firefox:

- **XPCOM (Cross Platform Component Object Model):** In order to record the triggered XHR callbacks, we instrumented the event queue in Firefox in `nsThread.cpp`. When events are taken from the queue for execution, we mark XHR `readystatechanged` events as well as button clicks initiated by the user.
- **Gecko (Layout engine):** We also need to instrument DOM API implementation of Firefox for recording updates made to DOM elements of interest. We achieved this by modifying various DOM class implementations like `nsGlobalWindow.cpp`, `DOMStorage.cpp`, etc.
- **SpiderMonkey (JavaScript engine):** Lastly, we instrumented the JavaScript interpreter for recording value manipulation on variables and objects and also to mark the start and end points of XHR callback execution. The former is achieved by instrumenting the JavaScript bytecode interpreter on `Interpreter.cpp` and `jsapi.cpp`.

Overall, our instrumentation is quite sparse and we believe can be easily migrated to another open-source browser such as Chromium. We have added a total of about 430 lines of instrumentation code to Firefox to collect our traces. A total of 12 files were modified.

Deployment strategies: The process of race detection is something that can be performed both online[3], as the application is running, as well as offline, as an *auditing* step. We envision that as part of beta-testing, traces from multiple users can be analyzed. Note that as we will highlight in Section 4, even relatively simple-looking sites can create long traces with a large number of events. At the same time, the number of events relevant to asynchrony and scheduling is relatively small. Our analysis for finding potential races is implemented as a linear pass over the trace. However, if desired, this is something that can be parallelized as well, by splitting longer traces to be analyzed on different machines.

3. FORMALIZATION

We find it convenient to represent the executions of event handlers, XHR callbacks, and portions of script code executed without pre-emption ("sequential blocks") as execution blocks with unique IDs. Other entries in an execution trace will also be assigned unique IDs as detailed later[4].

There is no universally accepted definition of the happens-before relation for web-based JavaScript code. We define the happens-before relation (denoted $\leftarrow$) not at the level of low-level memory accesses, but at the level of higher-level language constructs, based on causality information we abstract from JavaScript operational semantics as was illustrated in Figure 1. Within each uninterrupted execution block, trace entries are ordered by the program, and, therefore, happens-before order. We define and record a happens-before order between blocks such that if $id \leftarrow id'$ then all trace entries in id happen before those in id'. We order $id \leftarrow id'$ if id' appears *later* in the trace and one of the following hold:

[3]According to the data in Figure 6, our analysis is fast enough to be run online, so the beta testers only need to use a differently compiled version of the browser or perhaps a browser with a flag that they turn on. The results of such exploration can be centrally collected and communicated to the site developers.

[4]Note that these IDs are not to be confused with the statically-assigned numbers in Figure 1 as, for instance, the point in an execution where an XHR callback is registered and the point where it is executed are different.

1. Both blocks are sequential blocks. Sequential blocks are ordered by $\leftarrow$ in the order they appear in the trace because of the browser-imposed ordering.

2. Both blocks are event-handling blocks. In order to reflect the order of user interactions, event handling blocks are ordered by $\leftarrow$ in the order they occur in the trace to reflect the order of user interactions.

3. id' is an XHR callback, and the XHR `send` for id occurs within the block with ID id; this is because the XHR callback can only happen after the `send` operation.

3.1 Modeling and Analysis of Traces

In the rest of this section, we formally explain our race detection approach. The set of memory locations ("locations") manipulated by a JavaScript program is denoted by *Locs*, and the set of values they can take by *Val*. The value of an uninitialized locations is $\bot_{Val}$. We treat each field of each object as a unique, separate ordinary location for race-detection purposes. `document.cookie`, `sessionStorage` and `localStorage` are variables manipulated on a web browser by JavaScript programs. The type of each of these variables is a key-value store and they are persisted, i.e., stored on disk. KV denotes the set of these locations. To treat read and write accesses in the trace uniformly, we treat each (kv, ky) pair as a memory location with name $(kv, ky) \in Locs$. We use $\bot_{Val}$ to represent the value for keys not in the store. Accesses to distinct keys in a given $kv \in KV$, similarly to accesses to different memory locations, do not race with each other. The set of locations that represent DOM elements and are written to using the setter for the `innerHtml` property of a DOM element is denoted by $DOMElts \subseteq Locs$.

The value of each location v at each point in the execution is represented by a memory map named $\mathcal{V}$. $\mathcal{V}(v)$ consists of set of pairs of the form (vl, id) where $vl \in Val$ and $id \in IDs$. Intuitively, $(vl, id) \in \mathcal{V}(v)$ means that in this trace or a re-ordering of this trace that has the happens-before relation, v may have the value vl written to it by the block id. When processing a trace entry with ID id is a write access to v, we make critical use of the happens-before relationship computed up to that point. All entries in the memory map $\mathcal{V}(v)$ by trace entries with ID id' that happen before id are removed, representing the fact that they have been overwritten by a later access that happens after them. Other entries in $\mathcal{V}(v)$ are there due to "concurrent" execution blocks and are therefore not removed.

3.2 Defining Traces

Formally, a trace is a finite sequence of trace entries $\langle \lambda_0, \lambda_1, \lambda_2, \ldots \lambda_n \rangle$ of the following types:

- **XHR, event handling, and sequential blocks:** Trace entries $\lambda = CBBegin(id)$ and $\lambda = CBEnd(id)$ denote the beginning and end of the execution of the callback for the XHR with ID id. $\lambda = HandlerBegin(id)$ and $\lambda = HandlerEnd(id)$ do the same for event handler blocks, and $\lambda = SeqBegin(id)$ and $\lambda = SeqEnd(id)$ for sequential blocks.
- **Key, location, `innerHTML` accesses** $\lambda = keyWr(kv, ky, vl, varsRd, id)$ denotes the writing of the value vl for the key ky in the key-value store kv within the block with ID id. The value vl has been computed immediately prior to the write trace entry as the result of an expression over the memory locations $varsRd$. $\lambda = keyRm(kv, ky, id)$ denotes the removal of the value the key ky from kv within block id. $\lambda = varWrite(v, vl, id)$ denotes the writing of the value vl to the location v within block id. Finally, $\lambda = setHTML(hElt, hVal, id)$ denotes the setting of the `innerHtml` property of a DOM element $hElt$ to value $hVal$ within block id.
- **`POST`, XHR `send`:** $\lambda = post(url, id, id_{in}, vl, varsRd)$ is a `POST` request XHR call or Window with ID id and the call occurs within an execution block with ID id_{in}. The data posted vl is the computed result of an expression over the memory locations $varsRd$. $\lambda = xhrSend(id, id_{in})$ denotes an XHR `send` operation for the XHR object with ID id that takes place within an execution block with ID id_{in}. This `send` is a `GET` request.

3.3 Interpreting Traces

Given a trace $Trace = \langle \lambda_0, \lambda_1, \lambda_2, \ldots, \lambda_n \rangle$, our race detection algorithm analyzes it by processing it sequentially, one log entry at a time. The algorithm maintains analysis state represented by the tuple $\Sigma = (\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt})$. Here, $\mathcal{V}$ is the memory map. HB is the happens-before relation, which is a partial order over IDs. Whenever new elements are added to HB by a trace processing rule, the transitive closure of the relation is taken to obtain the resultant HB. HB is initially the empty relation. $\mathcal{P}$ is a list of pairs of the form (v, id) where the location v has been read while computing the value submitted by a XHR `POST` request with ID id. Finally, id_{Seq}, and id_{Evt} are the IDs of the last sequential block processed or the last event handling block processed by our algorithm, respectively, or $\bot$ if no such callback or block exists. Rules for updating the analysis state $\Sigma = (\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt})$ are given in Figure 4 and explained below.

Callbacks: CB-Begin and CB-End (not shown) keep track of the ID of the ongoing XHR callback block. Seq-Blk-Begin processes the log entry indicating the beginning of a sequential block with ID id and adds the pair $(id_{Seq}, id$ to the happens-before relation. Seq-Blk-End resets id_{Seq} to id. The rules ensure that the occurrence order of sequential blocks in the trace and their happens-before order coincide. Evt-Handler-Begin and Evt-Handler-End operate similarly to the corresponding Seq-Blk rules. The order of occurrence in the trace of the event handlers is the same as their happens-before order. Event handlers and sequential blocks are not ordered with respect to each other.

Location updates: Key-Write handles the case where key ky in the key-value map kv is updated. We declare a potentially-harmful race if $|Val(v)| > 1$ for any location v read while the new value for the key is being computed ($v \in varsRd$) indicating the potential for non-determinism. The rule computes $\mathcal{V}'(kv, ky)$ from $\mathcal{V}(kv, ky)$ by adding the pair (vl, id) and removing all pairs (vl', id') such that $id' \leftarrow id$. Key-Remove (not shown) writes the value $\bot_{Val}$ to the key ky. Write and Set-DOM are similar to Key-Write above, updating $\mathcal{V}$ for a location v or a DOM element $hElt$ by removing value-id pairs overwritten as dictated by the

Evt-Handler-Begin

$$\frac{\lambda = \mathit{HandlerBegin}(id) \qquad HB' = \mathit{TransClose}(HB \cup \{(id_{Evt}, id)\})}{(\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}, HB', \mathcal{P}, id_{Seq}, id_{Evt})}$$

Evt-Handler-End

$$\frac{\lambda = \mathit{HandlerEnd}(id) \qquad id'_{Evt} = id}{(\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id'_{Evt})}$$

Seq-Blk-Begin

$$\frac{\lambda = \mathit{SeqBegin}(id) \qquad HB' = \mathit{TransClose}(HB \cup \{(id_{Seq}, id)\})}{(\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}, HB', \mathcal{P}, id_{Seq}, id_{Evt})}$$

Seq-Blk-End

$$\frac{\lambda = \mathit{SeqEnd}(id) \qquad id'_{Seq} = id}{(\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}, HB, \mathcal{P}, id'_{Seq}, id_{Evt})}$$

XHR-Post

$$\frac{\begin{array}{c}\lambda = \mathit{post}(url, id, id_{in}, vl, \mathit{varsRd}) \\ \mathcal{P}' = \mathcal{P} \cup \{(v, id) | v \in \mathit{varsRd}\} \qquad HB' = HB \cup \{(id_{in}, id)\}\end{array}}{(\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}, HB', \mathcal{P}', id_{Seq}, id_{Evt})}$$

XHR-Send

$$\frac{\lambda = \mathit{xhrSend}(id, id_{in}) \qquad HB' = \mathit{TransClose}(HB \cup \{(id_{in}, id)\})}{(\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}, HB', \mathcal{P}, id_{Seq}, id_{Evt})}$$

CB-Begin

$$\frac{\lambda = \mathit{CBBegin}(id) \qquad id'_{CB} = id}{(\mathcal{V}, HB, \mathcal{P}, id_{CB}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}, HB, \mathcal{P}, id'_{CB}, id_{Seq}, id_{Evt})}$$

Key-Write

$$\frac{\begin{array}{c}\lambda = \mathit{keyWr}(kv, ky, vl, \mathit{varsRd}, id) \\ \mathcal{V}' = \mathcal{V}[(kv, ky) := \mathcal{V}(kv, ky) \cup \{(vl, id)\} \\ \qquad \setminus\{(vl', id') | id' \leftarrow id \text{ or } id' = id\}]\end{array}}{(\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}', HB, \mathcal{P}, id_{Seq}, id_{Evt})}$$

Key-Remove

$$\frac{\begin{array}{c}\lambda = \mathit{keyRm}(kv, ky, id) \\ \mathcal{V}' = \mathcal{V}[(kv, ky) := \mathcal{V}(kv, ky) \cup \{(\perp_{\mathcal{V}}, id)\} \\ \qquad \setminus\{(vl', id') | id' \leftarrow id \text{ or } id' = id\}]\end{array}}{(\mathcal{V}, HB, \mathcal{P}, id_{CB}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}', HB, \mathcal{P}, id_{CB}, id_{Seq}, id_{Evt})}$$

Write

$$\frac{\begin{array}{c}\lambda = \mathit{varWrite}(v, vl, id) \\ \mathcal{V}' = \mathcal{V}[(v) := \mathcal{V}(v) \cup \{(vl, id)\} \setminus \{(vl', id') | id' \leftarrow id \text{ or } id' = id\}]\end{array}}{(\mathcal{V}, HB, \mathcal{P}, id_{Seq}, id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}', HB, \mathcal{P}, id_{Seq}, id_{Evt})}$$

Set-DOM

$$\frac{\begin{array}{c}\lambda = \mathit{setHTML}(hElt, hVal), id) \\ \mathcal{V}' = \mathcal{V}[(hElt) := \mathcal{V}(hElt) \cup \{(vl, id)\} \\ \qquad \setminus\{(vl', id') | id' \leftarrow id \text{ or } id' = id\}]\end{array}}{(\mathcal{V}, HB, \mathcal{P}, id_{Seq}, , id_{Evt}) \xrightarrow{\lambda} (\mathcal{V}', HB, \mathcal{P}, id_{Seq}, id_{Evt})}$$

Figure 4: Trace analysis rules.

happens-before relation and adding the new value-id pair written by the current trace entry.

`POST`, `send`: The rule XHR-Post declares a potentially harmful race if $|\mathcal{V}(v)| > 1$ for any location $v \in \mathit{varsRd}$, since at least one location used in the computation of vl, the data posted, has the potential for non-determinism. XHR-SEND updates the happens-before relation. $\mathit{xhrSend}(id, id_{in})$ indicates that the send for the XHR with ID id has taken place in block id_{in}. $id_{in} \leftarrow id$. This rule and the fact that CB-Begin and CB-End do not modify the happens-before relation encode the fact that only chained XHR calls are ordered by $\leftarrow$ with respect to each other[5].

[5]Note that it is possible to define the happens-before relationship differently, for instance, declaring xhr_i to happen before xhr_j if the send entry for xhr_j appears later in the trace than the end of the callback for xhr_i. While such a definition may capture the happens-before relationship observed in a particular schedule more precisely, our definition narrows in on ordering relationships enforced by the JavaScript semantics and excludes those that may have taken place differently in different executions of the same program driven by the same user interaction.

3.4 Detecting Races

We say $\mathcal{V}(v)$ has *non-determinism potential on a location* v when the memory map contains at least two different values for v, i.e., when there are pairs (vl, id) and (vl', id') in $\mathcal{V}'(v)$, such that $vl \neq vl'$ and $id \neq id'$. Our algorithm declares a race on a location v while evaluating the Write, Set-DOM, and Key-Write rules if the memory map $\mathcal{V}'(v)$ computed by the rule has non-determinism potential on v. We also declare a race when evaluating the XHR-Post and Key-Write rules, if a variable read when computing the value posted or written, $v \in \mathit{varsRd}$, $\mathcal{V}(v)$ has non-determinism potential for v. Of these races, only the ones associated with Key-Write and XHR-Post rules are deemed to be harmful races.

Consider a prefix of the trace in a "quiet" state such that, at the end of the prefix, no XHR callback or execution block is in progress. Suppose that our algorithm has signaled non-determinism potential on a location v while processing a trace entry λ within the last block or callback with ID id in this prefix. Then a different re-ordering of the the execution blocks and/or XHR callbacks while leaving the happens-before relation in the prefix intact may result in a different final value for v, as explained next.

Let (vl, id) and (vl', id') be in $\mathit{Val}(v)$ such that $id' \neq id$ and $vl \neq vl'$. It must be the case that while both id' and id wrote to v, these two execution blocks are not ordered by the happens-before relationship. Otherwise, either (vl, id) or (vl', id') would have been removed from $\mathit{Val}(v)$, according to the update rules of our algorithm and how they use the happens-before relation in variable updates. Therefore, it is possible to modify the execution by delaying the execution of the block with ID id' until *after* the execution of the block with ID id. In this case, the final value of the location v or (kv, ky) would be different at the final state of the newly-obtained execution. This points to a potentially different result produced *purely* as a result of XHR callback scheduling non-determinism.

There are two sources of false positives in our race detection approach. The first is the assumption made in the previous paragraph while obtaining a new execution by delaying the execution of the block with ID id' past other blocks. The assumption is that control decisions made in the original execution based on data values written by the block id are *not modified* in a way that makes the reordered execution infeasible. In our empirical experience, such cases are rare and can be ruled out by inspection or replaying and validating the reordered execution. The second source is the check performed when a value is written to a persisted location or sent on the network. In these cases, if, while computing the written or sent value, a location with non-determinism is read, our algorithm signals a potentially harmful race. This approach is conservative, i.e., non-determinism in the value of one of the locations in an expression may not result in non-determinism in the value of the result. Even in cases where this is the cause of a false alarm, we believe that the potentially different data values flowing to a persisted output location may be of concern to programmers.

	XHR				Persistent state writes					Session state writes		DOM writes			
Web site	XHR `open`	XHR `send`	XHR Callbacks	Nested XHR Count	Cookie	Nested Cookie	Nested XHR `POST`s	`localStorage`	Nested `localStorage`	`sessionStorage`	Nested `session` `Storage`	Set `innerHtml`	Nested Set `innerHtml`	`INPUT` Element	Nested `INPUT`
edition.cnn.com	3	3	13	0	35	0	0	71	1	3	0	12,065	3	31	0
mlb.mlb.com	25	25	92	0	68	2	0	28	2	30	2	93	46	39	0
news.qq.com	8	8	28	0	17	0	0	8	0	0	0	153	10	64	2
wireless.att.com	22	22	72	3	79	4	0	46	32	11	2	902	847	60	5
aljazeera.net	12	12	32	2	24	0	0	6	0	0	0	1,095	29	46	7
bild.de	32	32	86	1	75	11	0	515	1	14	2	589	17	164	0
eltiempo.com	10	10	20	0	81	0	0	521	0	1	0	160	2	72	0
fedex.com.us	10	10	18	0	88	1	0	2	0	2	0	251	25	172	0
fujitv.co.jp	41	41	122	2	38	0	0	27	0	16	0	172	22	34	0
gazetta.it	22	22	67	4	115	15	0	68	0	19	0	119	24	52	0
girlsgogames.com	18	18	44	0	92	4	0	30	6	11	2	21	8	43	0
imdb.com	4	4	12	1	14	0	0	4	0	0	0	66	3	42	0
milliyet.com.tr	7	7	19	0	46	0	0	0	0	4	0	51	19	210	143
myvideo.de	20	20	87	0	99	0	0	115	0	8	0	86	0	23	0
nasa.gov	35	35	113	18	148	0	0	11	0	8	0	703	82	62	8
ntv.com.tr	10	10	14	0	58	1	0	55	0	293	0	112	3	66	0
nytimes.com	8	8	0	0	10	0	0	6	0	2	0	35	0	46	0
optimum.net	29	29	78	8	174	3	0	6	0	6	0	426	258	182	21
politico.com	27	27	65	9	231	47	0	36	18	1	0	110	19	248	10
premierleague.com	21	21	101	0	31	0	0	0	0	0	0	223	11	47	0
radikal.com.tr	13	11	20	0	111	0	0	33	0	2	0	85	5	59	0
sports.ru	28	28	90	1	48	0	0	84	0	0	0	128	58	52	1
sporx.com	5	5	0	0	33	0	0	31	0	0	0	49	0	38	0
tvguide.com	15	15	26	2	90	6	0	15	0	23	0	728	586	57	0
welt.de	23	23	44	1	112	0	0	366	0	4	0	546	19	68	0
zaobao.com	37	37	121	0	83	0	0	0	0	0	0	127	36	125	78

Figure 5: Characteristics of our benchmarks web sites.

4. EVALUATION

This section describes the experimental evaluation we have performed on a set of 26 complex pages.

Research Questions: Our goal has been to address the research questions listed below.

RQ1: How common are races on *persistent state* such as `document.cookie`, `localStorage`, and side-effects such as `POST` requests? We consider these to be the most harmful races there are in web applications.

RQ2: How common are races on `sessionState`? Session state is cleared on browser restarts. However, given that the browser is often not restarted for days if not weeks, data in `sessionState` can persist for long periods of time, if not permanently.

RQ3: How common are races on *transient state* such as memory locations and DOM elements? These are generally not the kind of errors that we deem to be highly problematic and, moreover, often these are not even observable by the user [10].

4.1 Experimental Setup

In our previous work [10], we used an instrumented browser to crawl all sites from Alexa's top 5,000 list. For our experiments here, we used 26 site of these sites that made heavy use of XHRs.

Site statistics: Figure 5 summarizes various aspects of the sites we picked. We specifically separate the number of operations within XHR callbacks, as these can lead to races. Columns 2–5 show the number of observed XHR `open` and `send` operations, executed XHR `onreadystatechange` callbacks, and the number of callbacks *within* other callbacks (so-called nested XHRs). Columns 6–11 focus on the use of persistent storage. Columns 6–7 show the number of writes to `document.cookie` as well as writes nested in an XHR callback. Column 8 shows the number of XHR `POST` operations in an XHR callback. Columns 9–10 give the number of writes to `localStorage` both in general and within XHR callbacks. Columns 11–12 give the same information for `sessionStorage`. Lastly, columns 13–16 give information about various forms of DOM manipulation such as setting `innerHtml` and changing the contents of `INPUT` elements; both totals and nested variants of these counts are provided.

Trace statistics: The workload used to collect these counts was simply loading the page and applying basic user interactions like button-link clicks. The counts for DOM manipulation are generally higher than those for `document.cookie`, `localStorage`, or `sessionStorage`. It is natural to expect more races on DOM elements as well, compared to more uncommonly used persistent elements. Figure 6 summarizes information about the traces we used for our analysis. The compression ratios range between 9.97 and 33.83. The percentage of time it takes to analyze a trace compared to the time to record a trace ranges between 3% and 32%.

Detection Time: Figure 6 shows the trace collection[6] and analysis time as a function of trace size. Analysis time grows approximately linearly with the size of the trace and is a fraction of the trace recording time shown in the second column.

[6]Sample traces can be found at `pastebin.com/MxF7ENPx`, `pastebin.com/JeSY4Fy8` and `pastebin.com/1pUsGUgb`.

Website	Trace Size (MB)	Compressed Trace Size (MB)	Browsing Time (sec)	Analysis Time (sec)
imdb.com	14.28	1.21	58	2
zaobao.com	30.75	1.40	38	6
news.qq.com	30.77	1.01	78	4
sporx.com	37.49	1.92	49	6
nytimes.com	40.58	1.26	47	8
fujitv.co.jp	45.79	1.39	72	8
girlsgogames.com	51.98	1.92	78	10
gazetta.it	93.02	3.07	182	17
radikal.com.tr	94.20	3.41	102	17
fedex.com.us	95.44	3.30	58	19
milliyet.com.tr	95.78	3.93	133	16
myvideo.de	104.82	4.38	121	18
welt.de	125.11	5.39	233	26
bild.de	138.47	5.74	119	22
mlb.mlb.com	142.81	4.61	81	21
wireless.att.com	151.22	4.47	87	24
tvguide.com	152.90	6.39	250	27
ntv.com.tr	162.09	6.50	158	27
aljazeera.net	169.87	6.72	93	28
sports.ru	209.68	8.33	257	29
edition.cnn.com	216.09	8.13	115	35
eltiempo.com	222.65	11.43	169	43
optimum.net	224.22	9.06	244	41
premierleague.com	264.08	8.16	244	47
politico.com	271.29	11.22	261	48
nasa.gov	407.76	40.91	425	56

Figure 6: Trace statistics and processing times.

Web site	`document.cookie`	`localStorage`	`sessionStorage`	`innerHtml`	`INPUT` element	Memory	`postMessage`
edition.cnn.com	0	0	0	0	0	12	0
mlb.mlb.com	1	0	0	0	0	28	0
news.qq.com	0	0	0	1	0	6	0
wireless.att.com	1	0	0	9	0	43	0
aljazeera.net	0	0	0	2	1	4	0
bild.de	0	0	0	0	0	58	0
eltiempo.com	0	0	0	0	0	9	0
fedex.com.us	1	0	0	1	0	10	0
fujitv.co.jp	0	0	0	1	0	13	0
gazetta.it	2	0	0	0	0	19	0
girlsgogames.com	2	1	0	0	0	0	0
imdb.com	0	0	0	0	0	8	0
milliyet.com.tr	0	0	3	0	0	5	0
nasa.gov	0	0	0	9	0	14	0
ntv.com.tr	0	0	0	0	0	13	0
optimum.net	2	0	0	10	3	22	0
politico.com	6	0	0	0	2	224	11
radikal.com.tr	0	0	0	1	0	11	0
sports.ru	0	0	0	0	0	17	0
tvguide.com	0	0	0	0	0	15	0
welt.de	0	0	0	0	0	19	2
zaobao.com	0	0	0	0	7	11	0
Totals	15	1	3	34	13	561	13
		Σ	**=19**	Σ	**=47**	Σ	**=574**

Figure 7: Races found by our analysis.

4.2 Detection Results

In this section we describe and analyze several representative races found with our approach.

Example 4 [`sessionStorage` in `milliyet.com.tr`.] In the case of `www.milliyet.com.tr`, the race on `sessionStorage` was caused by a shared variable `namespace` that is used for generating the key name. The variable is written at two different locations, one being an XHR callback that is executed using `jQuery.ajax` method. The execution steps are listed below and the relevant code is shown in Figure 8:

1. As a page loads, an XHR is created using the `jQuery.ajax` method (included in `www.milliyet.com.tr/D/j/base.js?v=20`) and sent to the server (lines 3–5);
2. once the response is received a user-defined callback is executed using `jQuery.ajax.done` (lines 19–22);
3. `namespace` variable is set to empty string at the end of `jQuery.ajax.done` method (line 21);
4. an external library is initialized by setting the `namespace` variable to `'__uv_'`(line 46);
5. `namespace` variable is subsequently used for generating a `sessionStorage` key (lines 43–45).

In the last step, the value of `namespace` is used for setting an item on the `sessionStorage` for recording user interactions with the web page. In this particular trace, we observe that multiple items are added to the `sessionStorage` using the `namespace` as a prefix (i.e. `__uv_autoprompt_disabled`, `__uv_r`). Any future read operations on the `sessionStorage` keys will depend on this prefix `namespace`. As the XHR callback can race with the `namespace` writes which sets the value for `namespace` to the empty string, it may result in a failed read operation. □

Example 5 [`document.cookie` in `gazetta.it`.] In this example of a `document.cookie` race from `www.gazetta.it`, the web site uses an application monitoring library (DynaTrace Real User Monitoring found at `dynatrace.com`) for recording and `POST`ing user actions and browsing experience to a remote server. Each `POST` request initiated by this library updates the key `dtCookie` of the `document.cookie` with the response value from the server. A race occurs when *multiple* XHR calls try to write the new response value to `document.cookie`. The execution steps are listed below, with line references from the code shown in Figure 9:

1. At the page load time, an XHR call is created for initializing the monitoring process (line 9);
2. the data is sent to a URL constructed from `dtCookie` value (lines 6–8);
3. `dtCookie` key is updated with the server's response (lines 35–37);
4. a new XHR is created at `onLoad` DOM event for posting loading time of the page (lines 11-13);
5. callbacks of each XHR try to write to the same `document.cookie` key `dtCookie` (lines 35–37).

The value of `dtCookie` key depends on the order of the XHR callbacks, causing possible issues with future `POST` operations, as the value is used for URL generation. □

4.3 False Positives

In looking for possible false positives, we have decided to focus our attention on both persistent races (19) and those on the DOM state (47). Out of the 66 races we investigated, we identified only *two cases* as false positives. These are `document.cookie` races in `fedex.com` and `optimum.net`.

Figure 10 shows the trace from `optimum.net`, with writes to the cookie key `fsr.s` at multiple locations (lines 6, 10,

```
<script>
  function MilGraphWithStatsV2(...){
    jQuery.ajax(method, url,
      // 1: www.milliyet.com.tr/D/j/base.js?v=20
      function (data) { //Process data });
      ...
  }
</script>
...a lot of text and images here...
<script>
  jQuery.extend({
   ajax: function (url, options){
     ...
     var xhr = new XmlHttpRequest();
     xhr.open(method, url)
     xhr.onreadystatechange = done;
     xhr.send(null);
     ...
     function done(...){
      // 2: www.milliyet.com.tr/D/j/base.js?v=20
      namespace = ""; // write to sessionState
     }
  }});
</script>
...
<script>
 // 3: widget.uservoice.com/uE6Mda0sQpbMbkAEhFaUig.js
 r.prototype.get = function(t) {
    if (this.storage) {
       var e = this.storage.getItem(this.makeKey(t));
       return e;
    }
 };
 r.prototype.set = function(t, e) {
   this.storage &&
      this.storage.setItem(this.makeKey(t),
    JSON.stringify(e))
 };
 r.prototype.remove = function(t) {
   this.storage &&
      this.storage.removeItem(this.makeKey(t))
 };
 r.prototype.makeKey = function(t) {
    return r.namespace + t
 };
 r.namespace = "__uv_";  // write to sessionState
</script>
```

Figure 8: sessionStorage manipulation in milliyet.com.tr. To save space, we remove unrelated lines of code.

```
<script>
// 1: rum-dytrc.gazzetta.it
//          /ajax/dtagent60_bjnprs3t_7082.js
 function sb(a, b) {
  var d;
  d = d + "?" + "dtCookie\x3d"
   + encodeURIComponent(v(Ua)) + ";"
   + encodeURIComponent(document.location.href);
  fb(cb, d, c, e)
 }
 E(q, "load", function(){
  fb(m, b.path, n, b.data);
 }
 function fb(a, b, c, d) {
   var h;
   h.onreadystatechange = function() {
    4 == a.readyState && (200 == a.status ?
     e(a.responseText) : db && B.sf &&
     eb.push({path: b,data: c}), a = m)
   }
   h.open("POST", b, c);
   h.send(d);
 }
 function e(a) {
  a = a && a.split("|");
  for (var b = 1; b < a.length; b++) {
    var c = a[b].indexOf("dtCookie\x3d");
    if (-1 < c) {
      y("dtCookie",
       decodeURIComponent(a[b].substr(c + 9)));
      break
    }
  }
 }
 function y(a, b) {
   document.cookie = a+"\x3d"+b
      +";path\x3d/" + w.domain;
 }
<\script>
```

Figure 9: Race on document.cookie manipulation in gazzetta.it. To save space, we remove unrelated lines of code.

and 22), one of which is within an XHR callback. In this case, fsr.s is used for collecting site statistics where the key value is updated by concatenating new values(lines 11–22). Although the XHR callback may happen in between two cookie writes on lines 5 and 6, causing a different value for the key, at the end of the execution the cookie key fsr.s will contain all the written values but in different orders. In this case, the program treats this value as a set and not a list, so, while we catch the non-determinism correctly, this is not really a bug in the program. The false positive in fedex.com is very similar, with a race is on cookie key s_sess, updated within an XHR callback.

Benign memory races: We would also like to highlight an example of a memory race that does not have persistent consequences. On radikal.com.tr, there is a global variable "duration" that is used to measuring time elapsed at different points throughout the execution on which there is a race between its update in sequential code and in an XHR callback. The value of this memory location does not propagate to any persisted state and cause non-determinism.

4.4 Discussion

Returning to the research questions in Section 4, we clearly see that according to Figure 7, races on persistent state (RQ 1) are quite uncommon, with only 19 for 26 sites. Races on session state (RQ 2) are also uncommon (only 3 races observed), in part because session state is used not as frequently as cookies (Figure 6). Lastly, races on transient state are considerably more frequent. While changes to HTML content may technically be races, one could argue that most of these are ephemeral. Indeed, the execution model of JavaScript-based web pages induces a great deal of non-determinism. This is because many if not most web pages are not the same across multiple reloads: the ads on the sides of the page change; content of pages often changes (consider a rapidly changing news site), sometimes the experience of the first visit to a page and subsequent ones is different because of cookies. The browser user has been trained not to expect much consistency and, when everything else fails, to reload the page. Additionally, the JavaScript execution model is extremely permissive: errors are "swallowed" by the runtime (the current event handler is terminated) and in many cases pages can survive exceptions and keep on running.

Our observations mesh well with the anecdotal experience of the user encountering any problem on the web: one only needs to reload the page for the problem to go away. In a sense, web programming is very forgiving. This is different from thread-related races in desktop applications and also data races in mobile apps [9, 5], where researchers are able to replicate races with obvious visual consequences (mangled or upside down images on the screen, for instance).

Lastly, ours is a *neighborhood exploration* technique. Given an execution trace, we explore the possible interleavings of callbacks. We record enough information to be *sound* for a given trace, but we choose to primarily focus our attention on write-write races on persistent state.

```
XHR [126F0800] Send
...
READ ID[235205312] = document :  JSObject
WRITE PROP ID[242159440] = cookie : JSString
Cookie [1020D800] Write "fsr.s=%7B%22v2%22%7D"   (1)
...
READ ID[235205312] = document :  JSObject
WRITE PROP ID[242159440] = cookie : JSString
Cookie [1020D800] Write "fsr.s=%7B%22v2%22%2C%7D"(2)
...
XHR [126F0800] Callback
BEGIN XHR_Callback
 ...
 READ ID[235205312] = document :  JSObject
 READ PROP ID[242159440] = cookie :  JSInteger — (3)
        "fsr.s=%7B%22v2%22%2C%7D"
 ...
 READ ID[235205312] = document :  JSObject
 WRITE PROP ID[242159440] = cookie : JSString
 Cookie [1020D800] Write
     "fsr.s=%7B%22v2%22%2C%22%3A1%7D" (4)
END XHR_Callback
```

Figure 10: Trace from `optimum.net` illustrating a false positive.

5. RELATED WORK

This section covers some of the recent work on finding races in JavaScript programs and asynchronous code.

Races in Asynchronous Programs: Hsiao *et al.* propose an approach to finding a subset of asynchronous races in Android apps, focusing on races that lead to use-after-free violations (i.e. uses of a freed pointer) [5]. While this tool also does offline analysis of a single execution trace, their focus is on computing an explicit happens-before relation, which they apply to the trace in order to find accesses that may lead to user-after-free possibilities. They employ some heuristics to minimize the possibility of false positives. We focus on data flow from multiple values to sensitive locations (like `document.cookie`), which naturally eliminates the need to explicitly reason about commutativity.

Maiya *et al.* [9] *et al.* focus on a systematic exploration of possible schedules using a UI explorer and reasoning about the obtained traces using a race detector. A precise model of the Android execution life cycle is key to avoiding false positives, although a large number of these remain.

Races in JavaScript: Zheng *et al.* [17] propose a static technique to detect potential races in JavaScript applications. More recently, Petrov *et al.* [11] and Raychev *et al.* [13] have observed the potential for asynchrony creating out-of-order execution and developed a notion of race conditions for Web applications written in JavaScript. In principle, race conditions can arise because of accesses to data shared among components of a Web page which are not ordered by proper synchronization, or, more formally, a happens-before relation. Of course, on a Web page, the entire DOM is (a giant blob of) global state, creating the potential for races.

Petrov *et al.* [11] define a happens-before relation for Web pages and generalize the notion of race conditions to take into account cases where, logically, there are unordered accesses to the same resource. The authors present a dynamic method for detecting races in a given execution of a Web page, explore similar executions that could potentially be racy, and, in later work [13] identify and filter out large sets of benign races. Hong *et al.* [4] propose a testing framework along with an execution model for asynchronous event handlers in JavaScript. Using defined execution model, the framework will execute a JavaScript application for collecting the executed asynchronous event handlers and then generates test cases for testing different orderings of these events. Later, the resulting DOM structure of each testcase is checked to the reference constructed by the initial execution. As discussed earlier, our work is distinguished from these studies by the fact that we only pursue race conditions that lead to non-determinism in persisted state or data sent to the server. Our choice of the happens-before relationship follows from this design decision and only records high-level causality relationships.

Mutlu *et al.* [10] advocate the notion of observable races, i.e. those that can be seen and visually distinguished by the end user. Our notion of persistent side effects is even stronger than that captured in this paper.

Program Analysis in JavaScript: A number of analyses have been propose for JavaScript in recent years; here we highlight only a handful. Additionally, several aspects of the language such as the use of `eval` [15, 6] and trying to understand JavaScript performance [12, 14]. Rozzle [7] proposes the idea of lightweight multi-execution in the context of a JavaScript engine, similar to our work. The goal of Rozzle is to expand the impact of malware detectors by increasing code coverage and thereby observing more, potentially malicious, code. In terms of techniques, Rozzle is probably the closest runtime exploration approach to the work described in this paper.

A project by Chugh *et al.* focuses on staged analysis of JavaScript and finding information flow violations in client-side code [1]. The Gatekeeper project [2, 3] proposes a points-to analysis together with a range of queries for security and reliability as well as support for incremental code loading. Gulfstream [3] is a successor of the Gatekeeper project whose focus is on incremental analysis and dynamic code loading. Sridharan *et al.* [16] presents a technique for tracking correlations between dynamically computed property names in JavaScript programs. Their technique allows them to reason precisely about properties that are copied from one object to another as is often the case in libraries such as jQuery.

Madsen *et al.* [8] proposes the idea of a *use analysis* for the purposes of call graph construction in JavaScript applications that use large frameworks and libraries. Their use analysis is combined with a points-to analysis for the rest of the application.

6. CONCLUSIONS

This paper proposes an alternative way of looking at what constitutes a race in web applications written in JavaScript. We advocate a focus on races that are caused by asynchronous callbacks and their order of arrival, primarily investigating races produced by the `XmlHttpRequest` (XHR) mechanism. Unlike prior work which concluded that there is ample potential for races in JavaScript, our findings suggest that given the forgiving nature of JavaScript applications, damaging, persistent races are considerably more rare.

Nevertheless, we propose a lightweight algorithm that explores different schedules in the "neighborhood" of a particular runtime trace. Our approach avoids the imprecision of static analysis and the combinatorial explosion and scalability issues of runtime schedule exploration. We find and investigate a total of 19 harmful races and 621 benign races in 26 web sites.

7. REPLICATION PACKAGE

This paper comes with an replication package designed to help other researchers working in this area. As part of our replication package, we submit

1. an instrumented version of the Firefox web browser, using which executions traces can be collected, and
2. our race detection tool which processes these traces for evaluation.

The Replication Packages Evaluation Committee found this submission to be satisfactory, and confirmed that our results can be reproduced. Our submission includes a virtual machine with the following contents[7]:

Race detector executable: Our race detection tool consists of a parser and a race detection module. A given trace file will be parsed and the state map and the happens-before relation at each point in the execution will be constructed. The executable accepts a trace path or directory of multiple traces and will output both the statistics of the corresponding trace(s) and detected race conditions at each state.

Instrumented Firefox executable: We instrumented the Firefox web browser in order to collect relevant information during execution of a web site. Our instrumentation spans over 12 source files with about 430 lines of instrumentation code. Most of the instrumentation is in the code for the JavaScript engine (SpiderMonkey) for recording memory manipulation operations. The event loop implementation of the web browser for marking asynchronous callbacks was also instrumented.

Collected traces of our benchmark web sites: We collected 26 individual traces for our benchmark web sites presented in the Section 4 using our instrumented Firefox browser.

We selected web sites from Alexa's top 5,000 list that made heavy use of XMLHttpRequests (XHR). For each web site, we conducted basic browsing actions (e.g., button clicks, navigating to new links) and collected the traces generated. The size of these traces varied from 14 MB to 400 MB, depending on the browsing time and content of the web sites.

Scripts for evaluation: We provide two batch scripts, `RunFSEAnalysis.bat` and `RunLastAnalysis.bat`, for automating the evaluation. These script will run our race detection over our benchmark traces and over the last collected trace with our instrumented web browser, respectively. The results containing the trace statistics, characteristics and detected races are persisted as `*.csv` files.

7.1 Evaluating Benchmark Web Sites

The replication package includes traces for collected each web site on our benchmark and a batch script (`RunFSEAnalysis.bat`) for running the race detection analysis on these traces for replicating the results presented in tables (Figure 5, Figure 6, and Figure 7). The script provided first runs our race detection mechanism on each web site trace and then presents the analysis results in separate `*.csv` files under the `FSE_Reports` directory:

- `LogCharacteristics.csv` contains characteristics (i.e. number of writes, number of XHRs etc.) for each analyzed trace(Figure 5).
- `LogStatistics.csv` contains statistics (i.e. trace size, analysis timing etc.) about each analyzed trace(Figure 6).
- `LogRaces.csv` contains a report summarizing the number of detected races on each analyzed trace (Figure 7).

Our race detection mechanism records both the statistics (see in Table 5) and detected race conditions (see in Table 7)) of the provided trace. After applying automatic race condition detection, we manually explored the reported races on persistent state. The analysis on the trace size, compressed trace size, and detection times is conducted by the batch script.

7.2 Collecting and Evaluating New Traces

Our replication package also includes an instrumented Firefox web browser for collecting new traces that can be analyzed with our race detection mechanism. Users can access the instrumented browser by executing the shortcut `FSE_Firefox` and use it in `Safe-Mode` for collecting new traces on web sites. Once the browser is terminated, the collected trace will be written to a file on disk. Users may experience some slowdown in the web browser due to the instrumentation and the use of a virtual machine.

The batch script provided, `RunLastAnalysis.bat`, first copies the last written trace file (named `%date%_%time%_log.txt`) to the `Last_Run_Reports` directory and then run the analysis on this trace. The results of the race detection analysis are then persistent as separate `*.csv` files described in the previous section along with the trace itself.

[7]The tools and instructions needed to replicate our results can be downloaded from `http://bit.ly/1O5GglJ`.

8. REFERENCES

[1] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for Javascript. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2009.

[2] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *Proceedings of the Usenix Security Symposium*, 2009.

[3] S. Guarnieri and B. Livshits. Gulfstream: Incremental static analysis for streaming JavaScript applications. In *Proceedings of the USENIX Conference on Web Application Development*, 2010.

[4] S. Hong, Y. Park, and M. Kim. Detecting concurrency errors in client-side JavaScript web applications. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, 2014.

[5] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn. Race detection for event-driven mobile applications. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2014.

[6] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *In Proceedings of the International Symposium on Software Testing and Analysis*, 2012.

[7] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[8] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2013.

[9] P. Maiya, A. Kanade, and R. Majumdar. Race detection for Android applications. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2014.

[10] E. Mutlu, S. Tasiran, and B. Livshits. I know it when I see it: Observable races in JavaScript applications. In *Proceedings of the Workshop on Dynamic Languages and Applications*, 2014.

[11] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby. Race detection for Web applications. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2012.

[12] P. Ratanaworabhan, B. Livshits, and B. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real Web applications. In *Proceedings of the USENIX Conference on Web Application Development*, June 2010.

[13] V. Raychev, M. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages*, 2013.

[14] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of JavaScript benchmarks. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, 2011.

[15] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *Proceedings of the European Conference on Object-oriented Programming*, 2011.

[16] M. Sridharan, J. Dolby, S. Chandra, M. Schaefer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *Proceedings of the European Conference on Object-oriented Programming*, 2012.

[17] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proceedings of the International Conference on World Wide Web*, 2011.