Race Detection for Web Applications

Boris Petrov
Sofia University
boris.petrov.petrov@gmail.com

Martin Vechev
ETH Zürich
martin.vechev@inf.ethz.ch

Manu Sridharan Julian Dolby
IBM T.J. Watson Research Center
{msridhar,dolby}@us.ibm.com

Abstract

Modern web pages are becoming increasingly full-featured, and this additional functionality often requires greater use of asynchrony. Unfortunately, this asynchrony can trigger unexpected concurrency errors, even though web page scripts are executed sequentially.

We present the first formulation of a *happens-before relation* for common web platform features. Developing this relation was a non-trivial task, due to complex feature interactions and browser differences. We also present a *logical* memory access model for web applications that abstracts away browser implementation details.

Based on the above, we implemented WEBRACER, the first dynamic race detector for web applications. WEBRACER is implemented atop the production-quality WebKit engine, enabling testing of full-featured web sites. WEBRACER can also simulate certain user actions, exposing more races.

We evaluated WEBRACER by testing a large set of Fortune 100 company web sites. We discovered many harmful races, and also gained insights into how developers handle asynchrony in practice.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Reliability; D.2.5 [*Software Engineering*]: Testing and Debugging—Monitors,Testing tools; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords concurrency, asynchrony, web, race detection, nondeterminism

1. Introduction

Modern web pages are increasingly becoming full-featured web *applications*, with rich user interfaces and significant client-side code and state. The web platform has significant advantages for application development, including quick deployment of updates, easier portability across desktops and mobile devices, and seamless client-server integration.

Due to the increasing popularity and complexity of web applications, there is a growing need for programming tools and reasoning techniques that match those for more mature platforms. This paper presents techniques to help developers make safer use of *asynchronous constructs* in the web platform. Web applications are making greater use of these constructs as their functionality becomes richer. For example, user interactions and completion of certain network requests can be processed in an asynchronous, event-driven style. Also, sites are making increasing use of delayed or asynchronous loading of JavaScript code itself, to speed initial page rendering and increase perceived responsiveness [24].

Use of asynchrony can lead to serious concurrency errors in web applications. Since the JavaScript code in web pages runs sequentially, there is less awareness of these concurrency errors than for languages like Java that have shared-memory multi-threading. Nevertheless, such errors can arise, due to non-determinism in event dispatch, network bandwidth, CPU speed, etc. (we give concrete examples in Section 2).

Such errors have caused serious bugs in real-world web applications. Developers at Mozilla noticed that many of the nondeterministic failures in their regression test suite were due to race conditions in the unit test inputs [20], leading to documentation on how to avoid such problems [19]. The Hotmail email service was broken in the Firefox web browser for some time due to a race, with the bug causing a loss of message content.[1] Race conditions leading to data loss have been discovered in this paper and have also been reported in previous work [25].

To clarify the behavior of asynchrony in web applications, we present the first formulation of a *happens-before relation* [16] for the most commonly-used JavaScript and HTML features. While various types of web-application concurrency errors have been discussed previously [15, 19, 24, 25], we are unaware of any discussion that formulates these issues over a common model. Furthermore, while there are ongoing efforts to specify the web platform more carefully [10], defining a useful happens-before relation is still non-trivial, due to complex interactions between JavaScript and standard HTML features and browser deviations from the specification. The happens-before relation presented here was developed based on an in-depth study of relevant specifications, browser behaviors, and how constructs are used in practice.

Another key contribution of our work is a model of which *logical* memory locations are being accessed by various web platform features, necessary for building tools like a race detector. Typically, memory accesses can simply be defined as those reads and writes occurring at the machine or virtual-machine level. However, for web applications, there is no obvious definition of machine-level accesses, as operations may access JavaScript heap locations, browser-specific native data structures, or both. Our model of logical memory locations enables reasoning about these memory-access operations in a browser-independent manner.

Based on our models of the happens-before relation and logical memory locations, we implemented WEBRACER, a dynamic race detector for web applications.[2] The dynamic approach lets WEB-

PLDI'12, June 11–16, 2012, Beijing, China.

[1] `https://bugzilla.mozilla.org/show_bug.cgi?id=538892`

[2] Apart from dynamic race detection, our models are also a suitable basis for other concurrency analyses, e.g., static race detection or atomicity checking.

RACER precisely handle many complex features of web applications that would be difficult to handle with static analysis alone. The JavaScript language has many difficult-to-analyze constructs, like prototype chains and `eval`, that are used frequently in real-world applications [22]. WEBRACER can simply observe the relevant effects of these constructs, side-stepping difficult static analysis issues. Further, WEBRACER precisely handles interactions between JavaScript and HTML via the Document Object Model (DOM) data structure, a tree representation of the HTML often queried via string matching of node identifiers. WEBRACER is able to find races at the level of concrete DOM tree nodes (i.e., individual HTML elements) by observing accesses to memory addresses. WEBRACER is also able to simulate certain user interactions, exposing more races.

We implemented WEBRACER atop WebKit [3], a robust rendering engine used in many production browsers (e.g., Safari[3] and Google Chrome[4]). Using a production engine allowed for WEBRACER to be tested on full-featured, real-world web sites. In an experimental evaluation, we ran our tool on a large set of Fortune 100 company web sites and detected thousands of race conditions. A manual inspection of a subset of reported races showed that many of them reflected real bugs. We also gained insights into how web developers manage asynchrony in practice, applicable to future tools. Note that finding real bugs on deployed web sites was quite a challenging test for WEBRACER due to our unfamiliarity with the sites' code and frequent code obfuscation; we expect WEBRACER to be even more effective for a developer debugging her own site.

This paper makes the following contributions:

- We give the first formulation of a happens-before relation to capture the asynchronous behaviors of most commonly-used web platform constructs.
- We define a model of how operations in web applications access logical memory locations, independent of browser implementation details.
- We present WEBRACER, a dynamic race detector based on our models and built on the production-quality WebKit engine.
- We describe an evaluation of WEBRACER on a large set of production web sites, in which we discovered many harmful races and gained insights into how asynchronous constructs are used in practice.

This paper is organized as follows. Section 2 gives examples of web site concurrency errors. Section 3 presents our happens-before relation. Section 4 shows our logical memory access model. Section 5 describes our race detection algorithm and implementation. Section 6 presents our evaluation. Section 7 discusses limitations. Section 8 discusses related work, and Section 9 concludes.

2. Motivation

In this section, we explain in more detail how data races can arise in web applications, give several motivating examples illustrating possible types of races, and show the potential harm caused by such races. We also use the examples to illustrate the happens-before relationships and logical memory accesses that our model must include.

2.1 Sources of Races

For our purposes, the execution of a web application can be roughly seen as consisting of two types of activities: (1) conversion of HTML into a DOM tree [1] and (2) execution of scripts containing JavaScript code. In most modern browsers, these activities are always interleaved in a single thread of execution, prohibiting many types of true concurrency (e.g., two distinct scripts cannot execute concurrently).[5] So, the traditional definition of a data race involving two unordered memory accesses (one of which must be a write) from distinct execution threads does not apply directly.

```
<script>x = 1;</script>
<iframe src="a.html" />
<iframe src="b.html" />

<!-- a.html -->
<script>x = 2;</script>

<!-- b.html -->
<script>alert(x);</script>
```

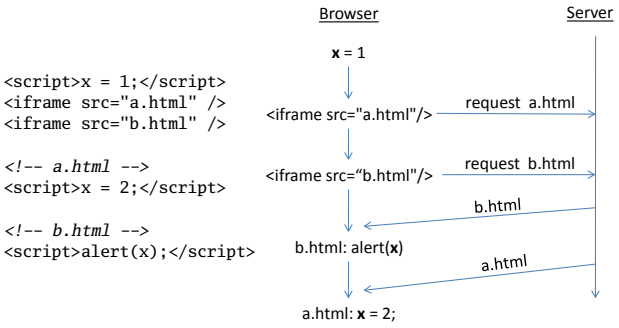

Figure 1. On the left is a simple example containing a race on **x**. On the right is an execution of the code showing the race on **x**.

Instead, races in web application arise from environmental asynchrony, typically triggered via event dispatch. The web platform employs an event-based programming model to handle a variety of external events—pages may register handler code to be executed after user interactions (e.g., a mouse click) or the completion of some page loading operation, etc. Scripts may also explicitly queue functions to execute after some amount of time via the `setTimeout` or `setInterval` methods [2]. These event handler functions may execute in a non-deterministic order due to a variety of factors, e.g., variation in network bandwidth, CPU resources, or the timing of user input events.

Asynchronous events often lead to race conditions in conjunction with a web browser's rendering of a *partially-loaded* page. During the page load process, modern browsers aggressively attempt to render the HTML and JavaScript code downloaded so far, thereby improving perceived browser performance. When a partially-loaded page is rendered, user interactions may be interleaved with the remainder of the page load process in an unexpected manner, leading to race conditions. Furthermore, web application authors often exploit partial page rendering by deliberately delaying the download and execution of certain script code. In such cases, the page appears to load quickly, but some code required for user interaction may not yet be loaded, again leading to races.

We shall now illustrate these issues with examples of four types of data races. We start with standard data races on JavaScript memory locations, similar to those seen in other languages like Java. We then illustrate three types of races which are more specific to the web platform: HTML races, function races, and event dispatch races. These race types are worth distinguishing since the racing accesses do not always appear as standard memory accesses in the JavaScript code.

2.2 Variable Races

As in many other languages, web applications may have data races on program variables, i.e., JavaScript memory locations. A very simple example appears on the left side in Fig. 1. Here, a variable `x` is set to `1` in the global scope. Then, two `iframe` elements are created;

[3] `http://www.apple.com/safari/`

[4] `http://www.google.com/chrome`

[5] The Opera web browser (`http://www.opera.com`) may execute scripts from different frames concurrently; we do not consider this case in the current work, as all other major browsers avoid such behavior.

```
<input type="text" id="depart" />
...
<script type="text/javascript">
// add a hint to the box
document.getElementById("depart").value =
  "City of Departure";
// code to remove hint when user clicks
...
</script>
```

Figure 2. Southwest: A data race on a form field value.

```
<script type="text/javascript">
function show(emailTo,EmailC) {
  ...
  EmailC.value = emailTo;
  v = $get('dw');
  v.style.display = "block";
}
</script>

....

<a href=
"javascript:show('x@x.com', a)">
    Send Email
</a>
...

<div id=dw style="display:none">
  // HTML form to send email
</div>
```

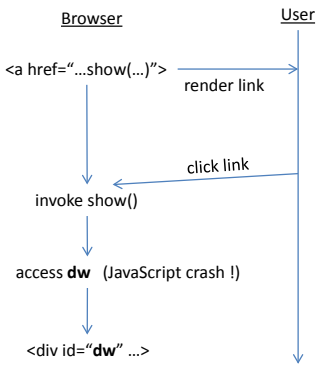

Figure 3. On the left is an example containing an HTML race on **dw**. On the right is an execution of the code showing the race on HTML element **dw** that causes a JavaScript error (crash).

an `iframe` contains the content of some other HTML file, loaded asynchronously. In this case, the `iframe`s cause `a.html` and `b.html` to be loaded asynchronously. The script in `b.html` may display `1` or `2` depending on when the script in `a.html` is executed, a clear data race on variable `x`. The trace shown on the right in Fig. 1 illustrates the case where `b.html` displays `1`.

Note that in Fig. 1, the first write `x = 1` does *not* race with the write `x = 2` in `a.html`, since the first script will always execute before the `iframe`s are loaded. Our happens-before relation accurately captures such orderings between HTML element parsing and script execution.

Data races may also occur on properties of a DOM node (representing an HTML element), potentially making them more directly visible (and annoying) to the user. Consider the example of Fig. 2, a simplified version of a real bug discovered in the `southwest.com` web site.[6] First, an `input` element is created in which the user can type their departure city for a flight search. Later in the page, a script sets the value in the input box to "City of Departure", as a hint to the user, and adds code to make the hint text disappear when the user clicks in the box (not shown). Here, the non-determinism stems from partial page rendering: the user may see and interact with the input text box *before* the script loads and runs, and if this occurs, the script will simply overwrite any text that the user has entered! Our race detection tool is able to discover this type of bug *automatically* by simulating certain user interactions (details in Section 5.2.2).

2.3 HTML races

Certain web application data races are more unique to the semantics of the web platform and JavaScript. An *HTML race* occurs when an access of a DOM node representing an HTML element may occur before or after its creation. Consider the left side of Fig. 3, based on code with a race from `valero.com` discovered by our tool. Here, the `div` at the end of the example with id `dw` holds an HTML form for sending email, hidden by default (since the value of the `style` attribute is `display:none`). When the user clicks the "Send Email" link, the `show()` JavaScript function executes and the style of `dw` is changed, making the form appear. (The `$get` function employed by `show()` is essentially equivalent to the `document.getElementById()` function.)

Here, the problem arises if the user clicks the "Send Email" link *before* the `dw` element has loaded, as shown in the trace in the right part of Fig. 3. In this case, the `show()` function will attempt to set the style of a non-existent element, leading to an exception being thrown and termination of JavaScript execution. Web browsers are designed to hide many "bad" JavaScript behaviors, and in this case, the JavaScript crash will not be shown to the user—the link will simply appear to do nothing when clicked, and subsequent scripts will continue to be executed. These hidden crashes can mask more serious problems, as mutations to global JavaScript state preceding the crashes persist, possibly leaving objects in an inconsistent state. In Fig. 3, the effect of the statement `EmailC.value = emailTo` in `show()` will remain even if the subsequent statement accessing `dw` crashes, potentially affecting execution of subsequent scripts on the page.

Note that for this case, it is not obvious which "memory location" the operations race upon. Within a browser implementation, the concrete memory location(s) involved may depend on how exactly the DOM data structure is implemented. In our model, we define a logical HTML element location l for the `dw` element, with the JavaScript `$get` call reading l and the browser's parsing of the `div` node writing l. In this manner, the data race can be modeled independent of browser implementations.

```
<iframe id="i" src="bug413310-subframe.html"
  onload="setTimeout(doNextStep, 20)">
...
<script type="text/javascript">
...
function doNextStep() {...}
</script>
```

Figure 4. An example of a function race.

2.4 Function races

Similarly to an HTML race, a *function race* occurs when an invocation of function f may occur before or after the parsing of f.[7] Fig. 4 gives an example of a function race, extracted from a Mozilla Firefox unit test that was failing non-deterministically. Here, we have an `iframe` whose `onload` handler (executed after the `src` HTML file is loaded) uses `setTimeout()` to perform a delayed invocation of `doNextStep()`, declared in the later `script` tag. The problem is that even with the 20ms delay, `doNextStep()` may be invoked before its declaring script is loaded if the `iframe`'s HTML loads too fast. Invoking a non-existent function in JavaScript causes an exception, which in this case would cause the corresponding unit test to fail. In general, these exceptions may lead to the same types of problems described in Section 2.3 with HTML races, due to JavaScript code possibly being terminated in an inconsistent state. Here, the race can be fixed by moving the `script` element above the

[6] This bug appears to have been fixed in the latest version of the site.

[7] One site we ran WEBRACER on contained this code comment, indicating awareness of function races:
`/* Duplicated the JS function to get rid of race condition happening in the dashboard page. */`
We do not endorse their solution.

```
<iframe id="i" src="a.html" />
...
<script>
document.getElementById("i").onload = function() {...}
</script>
```

Figure 5. An example of an event dispatch race.

`iframe`, in which case our happens-before relation would show that `doNextStep()` is always parsed before being invoked.

2.5 Event dispatch races

An *event dispatch race* occurs when an event may fire before or after some handler for that event is added. Fig. 5 gives a small example of such a race. Here, the `onload` handler for an `iframe` is set in a separate script, rather than directly in the `iframe` tag via the `onload` attribute. Third-party scripts often add event handlers in this way, since they cannot modify attributes in the HTML source directly. With this example, it is possible that the `iframe`'s load event will fire even before the script executes (if the `iframe` loads very quickly), in which case the installed `onload` handler will never run.

Note that for this example, one racing "access" is the read of the `iframe`'s `onload` attribute by the browser when the `iframe`'s `load` event is dispatched—this read is not explicit in the HTML or JavaScript code. Our model of happens-before and logical memory can both expose this race and also show that there is no race if the `iframe`'s `onload` attribute is set in the tag itself.

3. Happens-Before

In this section we formulate a happens-before relation that captures key ordering constraints for the most common web-platform features. First, we give a brief background on the relevant HTML and script constructs. Then, we define *operations* that comprise (atomic) execution. Finally, we define the *happens-before* relation between operations.

We note that defining the happens-before relation can be quite challenging because relevant specifications can be vague and sometimes browsers differ on how they implement the specification. This section represents our best effort to define a happens-before relation that agrees with both the specification [10] and how most major browsers work. In cases, where the specification was unclear or behavior differed significantly across browsers, we erred on the side of not adding happens-before edges in order to not miss races.

3.1 Background

First, we give some brief, informal background on the web platform constructs discussed in this section. Due to space constraints, we cannot give a complete treatment of all relevant features; the reader should consult online documentation and specifications for further details [10, 17].

HTML An HTML page consists of a tree of *elements*. Each element is typically delimited with an opening and a closing tag, e.g., `<p>...</p>` for a paragraph element. Within an HTML page, we say that element e_1 *precedes* element e_2 if e_1's opening tag appears syntactically earlier than e_2's opening tag. For instance, in the example below, element `a` precedes elements `b` and `c`, and element `b` precedes element `c`:

```
<div id="a">
  <div id="b"></div>
</div>
<div id="c"></div>
```

A *static* HTML element is declared syntactically in the page; elements may also be inserted by scripts, as discussed below.

Scripts and the DOM JavaScript code is added to a page via `<script>` elements. A static script element is *inline* if its code is declared in its body, e.g., `<script>x = 10;</script>`. Otherwise, a script element is *external*, and its code resides in a file specified via the `src` attribute, e.g., `<script src="code.js"></script>`. We elide a detailed description of the JavaScript programming language for space; see external resources for further details [6, 17].

The *Document Object Model (DOM) tree* [1] is a parsed representation of a page's elements that gets rendered by the web browser and possibly accessed or mutated by scripts. Each node in the DOM tree has *attributes* to hold meta-data, including attributes that mirror those seen in the corresponding HTML tag (e.g., a node representing a `<script>` element may have a `src` attribute).[8] Scripts can add or remove nodes from the DOM, leading the web browser to update its page rendering. Scripts may also insert new script nodes into the DOM to make the browser load and execute new code (possibly asynchronously); we say a script added in this manner is *script-inserted*. As with static `<script>` elements, a script-inserted script is *inline* if its code is present as a child node and *external* if the code location is in its `src` attribute.

Asynchronous and deferred scripts Both static script elements and script-inserted scripts may be declared as *asynchronous* or *deferred* via attributes. Intuitively, a deferred script should run after all static HTML elements have been parsed, while an asynchronous script may run at any time (we capture the constraints more precisely in Section 3.3). An asynchronous script has a boolean `async` attribute with value `true`, and a deferred script has a similar `defer` attribute, e.g.:

```
<script src="code1.js" async="true"></script>
<script src="code2.js" defer="true"></script>
```

Asynchronous and deferred scripts must be external (i.e., their `src` attribute must be set), and a script cannot be both asynchronous and deferred. An external script that is neither asynchronous nor deferred is a *synchronous* script.

Frames HTML pages may be embedded in *inline frames* in other pages via the `<iframe>` tag, e.g.:

```
<iframe src="nested.html"></iframe>
```

The HTML in an inline frame is loaded asynchronously, making `iframe`s interesting from a happens-before perspective. The root HTML page and each (transitive) inline frame has its own *window* object, and each window has an associated *document* object, essentially the root of the corresponding DOM tree.

Events and Handlers Web applications must be written in an event-driven style, registering handler scripts to be executed when various types of events occur. Events are dispatched, e.g., for user interactions (clicks, typing, etc.) and the completion of loading various elements (images, scripts, etc.). Every event has a *target* denoting the object upon it was dispatched (e.g., for a `click` event, the DOM node for the `button` that was clicked).

Asynchronous networks requests (so-called "AJAX" requests) are made by invoking `send()` on some `XmlHttpRequest` object o. At various stages of the request (including completion), the `readystatechange` event is dispatched with target o.

For simplicity, we ignore two features of event dispatch in this section (though they are fully handled by our implementation):

1. Inline event dispatch, where a script explicitly fires an event via a method call.
2. Propagation of events through a DOM tree, via "capturing" and "bubbling" [5].

These features, and corresponding minor modifications to the happens-before rules, are discussed further in Appendix A.

[8] Strictly, elements have *content attributes*, while DOM nodes have *IDL attributes*; we call both attributes here.

DOM content and window load Two events of particular interest are the `DOMContentLoaded` event on a document, indicating (roughly) that the static HTML for the document has been parsed, and the `load` event on a window object, fired after resources like images and inline frames have fully loaded. Scripts very often register handlers on these events to perform additional computation once other resources have been loaded, so capturing their happens-before relationships precisely is important.

Timed execution The `setTimeout` and `setInterval` functions can be used to perform delayed execution of some script code. A call `setTimeout(`f`,`i`)` causes f to be executed i milliseconds or later. `setInterval(`f`,`i`)` is similar, but it executes f every i milliseconds. Due to their timing-dependent behavior, capturing happens-before for these functions is crucial.

3.2 Operations

Next, we define the kinds of *operations* we consider in this work. These operations will be used to specify the happens-before relation in Section 3.3. Each operation has a unique identifier taken from the set $OpId$.

Strictly speaking, we have only two types of atomic operations during web page loading: (1) parsing of HTML or (2) execution of script code. We write $parse(E)$ for the operation that parses a static HTML element E. For convenience, we separate script execution operations into several types:

- $exe(E)$: the operation executing the source in a `script` element E (either static or script-inserted).
- The execution of an event handler due to an event dispatch.
- $cb(E)$: the execution of the callback script E resulting from a `setTimeout(`E`, _)` call.
- $cb^i(E)$: the execution of the i'th invocation ($i \geq 0$) of a callback script E resulting from a `setInterval(`E`, _)` call.

We also introduce some helper functions related to operations:

- $create(E)$ denotes the operation that inserts an element E into a document (i.e., a DOM tree). If E is a static HTML element, then $create(E) = parse(E)$. Otherwise, $create(E)$ is the operation associated with the script that inserts E.
- $disp_i(E, T)$ denotes the set of operations that execute all event handlers for the i^{th} dispatch of event E at target T.
- $ld(T)$ denotes $disp_0($`load`$, T)$, if T has a `load` event.
- $dcl(D)$ denotes $disp_0($`DOMContentLoaded`$, D)$ for a document D.

3.3 Building the Happens-Before

Here, we give a complete definition of our happens-before relation for the web features described in Section 3.1. As stated earlier, we developed this relation based both on studying the specifications [5, 10] and common browser behaviors, aiming to only introduce rules where all of them mostly agreed (with fewer happens-before edges, more possible races are exposed).

The happens-before relation, denoted $\prec$, is a binary relation on operation identifiers, i.e. $\prec \subseteq OpId \times OpId$. As a shortcut we use $A \prec B$ to mean $(A, B) \in \prec$ and $A \not\prec B$ to mean $(A, B) \notin \prec$. When we say that that two operations are in the happens-before, we mean the identifiers of these operations.

Sometimes we need to define a happens-before between an operation A and all operations found in a set B. In that case, we overload $A \prec B$ to mean $\forall (a, b) \in \{A\} \times B.\ a \prec b$. The definitions of $A \prec B$ when A is a set and B is an operation or when both A and B are sets is similar. In the rules below, only the identifiers $disp_0$, ld and dcl represent sets.

We group our rules for constructing the happens-before in roughly the same order as the order in which the corresponding features are described in Section 3.1.

Static HTML Elements in static HTML are essentially processed in syntactic order, i.e.:

1. Let E_1 and E_2 be two static HTML elements in the same document, such that E_1 precedes E_2 (see Section 3.1). Then:
   (a) $parse(E_1) \prec parse(E_2)$.
   (b) if E_1 is an inline script, $exe(E_1) \prec parse(E_2)$.
   (c) if E_1 is a synchronous script, $ld(E_1) \prec parse(E_2)$.

Script Parsing, Execution, and Loading The following basic rules govern all script elements E:

2. $create(E) \prec exe(E)$
3. $exe(E) \prec ld(E)$ (except for inline scripts, which have no `load` event).

Note that script-inserted inline scripts execute synchronously and their code does *not* execute as part of a new operation.[9]

Asynchronous and Deferred Scripts Static deferred scripts execute in syntactic order after the DOM content has been loaded, captured with the following rules.

4. Let E be any element in a document D such that $create(E) \prec dcl(D)$ and let S be a static deferred script element in D. Then $create(E) \prec exe(S)$.
5. If E_1 and E_2 are static deferred script elements, and E_1 precedes E_2, then $ld(E_1) \prec exe(E_2)$.

Asynchronous scripts and external script-inserted scripts may execute in any order. Apart from rules 2 and 3, such a script is only governed by rule 15, relating its `load` event to that of the containing window.

Inner Frames The HTML nested in an `iframe` element I loads asynchronously, with the following constraints:

6. For any element E in the nested document for I, $create(I) \prec create(E)$.
7. I's `load` event fires after the `load` event for the nested window W_I, i.e., $ld(W_I) \prec ld(I)$.

Event Handlers Some basic rules apply to event handler execution. Consider $A \in disp_i(e, T)$ for some $i \geq 0$, event e, and target T.

8. The target must have been created previously: $create(T) \prec A$.
9. For any $B \in disp_j(e, T)$, where $0 \leq j < i$, $B \prec A$.

We also have the following rule for AJAX requests:

10. Let A be the operation invoking `send()` on an `XmlHttpRequest` object T. Then, $A \prec disp_0($`readystatechange`$, T)$.

DOM Content and Window Load These rules define happens-before relationships for the `DOMContentLoaded` event on a document and the `load` event on a window (in addition to rule 7). We begin with a basic rule relating the two:

11. Let D be the document of a window W. Then $dcl(D) \prec ld(W)$.

The following rules indicate which operations must happen before the `DOMContentLoaded` event for a document D:

[9] Recent browser versions adhere to this rule, but older versions of Firefox did not [23].

12. Let E be a static HTML element in D. Then $parse(E) \prec dcl(D)$.
13. Let E be a static inline script element in D. Then $exe(E) \prec dcl(D)$.
14. Let E be a static synchronous or deferred script element in D. Then $ld(E) \prec dcl(D)$.

Finally, this rule shows which events must precede the `load` event for a window W:

15. Let E be an element in the document of W s.t. $create(E) \prec ld(W)$ holds and E has a `load` event (e.g., an `img` or `script` element). Then $ld(E) \prec ld(W)$.

Note that due to rule 3, rule 15 also relates the execution of scripts and the window `load` event.

Timed Execution The following two rules govern `setTimeout` and `setInterval` executions:

16. Let A be an operation which calls `setTimeout(`B`, _)`. Then $A \prec cb(B)$.
17. Let A be an operation which calls `setInterval(`B`, _)`. Then $A \prec cb^0(B)$ and $\forall i \geq 0. cb^i(B) \prec cb^{i+1}(B)$.

Finally, we note that the happens-before relation is transitive: if $A \prec B$ and $B \prec C$, then $A \prec C$. Hence, the final relation $\prec$ is built by taking the transitive closure of all pairs given above.

4. Memory Accesses

In this section, we describe the shared *memory accesses* that an operation can perform. The notion of memory access is complicated by the fact that the web platform has no natural definition of "machine-level" accesses, as common operations manipulate both JavaScript heap locations and browser-internal data structures (e.g., DOM operations). Here, in addition to the usual JavaScript variables, we identify HTML elements in the DOM and event handlers as key *logical* locations accessed by operations. We define accesses to these locations in a browser-independent manner, easing high-level reasoning. We discuss each type of location in turn.

4.1 Accesses on Variables

Let $JSVar$ be the set of JavaScript variables that could potentially be shared among different operations. Such variables may include:

- Local variables (which could be shared between different operations via a closure).
- Object properties (instance fields and array element).
- Global variables (which are technically properties of a "global object").

By $JSVar$ we mean the set of concrete runtime memory addresses corresponding to these JavaScript variables. Reads and writes of these addresses are potential shared memory accesses.

Functions We treat a declaration of a function named F (i.e. not a lambda function) in JavaScript scope S as a write of an anonymous function with F's body to a local variable named F, where the local variable assignment is placed at the beginning of scope S, in accordance with JavaScript semantics. For example, the following:

```
{ // scope S
   some_statements A;
   function foo() { some_statements B; }
   some_statements C;
   function bar() { some_statements D; }
}
```

is treated as:

```
{ // scope S
   var foo = function() { some_statements B; };
   var bar = function() { some_statements D; };
   some_statements A;
   some_statements C;
}
```

Additional Cases The following accesses are modeled as writes to properties of DOM objects in the JavaScript heap:

- Adding/removing a child element B to/from an element A (whether statically or dynamically) is considered a write to B's `parentNode` property and a write to A's `childNodes[i]` property, where i is the index of B in the `childNodes` list.
- Modification of an HTML form element is treated as a write to the corresponding DOM node attribute. For example, the user typing into an `input` or `textbox` element is considered a write to the element's `value` attribute, clicking a `checkbox` writes its `checked` property, etc.

4.2 Accesses on HTML Elements

Let $HElem$ denotes the set of HTML elements. Then:

Write Accesses The following write to an HTML element e:

- Inserting e (either via static parsing or dynamic JavaScript insertion) into a document. Dynamic insertion of an HTML element also dynamically inserts all of its child elements. The `appendChild` and `insertBefore` functions in JavaScript are examples of ways to dynamically insert elements.
- Removing e (dynamically via JavaScript) from a document (which also removes its child elements). The `removeChild` function in JavaScript is an example of a way to dynamically remove an element.

Read Accesses JavaScript code can perform a logical read of an HTML element e via accessor methods or direct reads. Examples include:
`document.getElementById`, `document.body`,
`document.getElementsByName`, `document.forms[i]`,
`document.getElementsByTagName`, `document.images[i]`,
`document.childNodes[i]`, `document.anchors[i]`,
`document.links[i]`, `document.scripts[i]`

4.3 Accesses on Event Handlers

The combination of a target element el, event e and event handler h defines a logical event handler location $(el, e, h) \in Eloc$. Note that by having h in the logical location instead of only el and e we allow accesses that manipulate disjoint handlers for the same event e to not interfere.

Write Accesses The following accesses write an event handler location:

- Parsing of an element with an event handler content attribute [10, Section 6.1.6.1], for example:
  `<img id="g" onload="doWorkA()"></img>`
- Writing the event handler attribute of an element, for example:
  `document.getElementById("g").onload = "doWorkB()"`
- Invoking the `addEventListener` function on an element
- Invoking the `removeEventListener` function on an element

Read Accesses An event handler location (el, e, h) is read when executing event handler h due to an event dispatch of event e with current target el. An event dispatch could be initiated by a user (e.g. clicking a button) or programmatically (e.g. calling `el.focus()` to dispatch a `focus` event on element `el`).

5. Race Detection

In this section, we define the notion of a data race based on our happens-before relation (Section 3) and memory access model (Section 4), describe our race detector, and finally discuss WEBRACER, our WebKit-based race detector implementation.

5.1 Race Detector

We next define what a race is, and then we present our dynamic race detector.

Definition of a Race From the definition of happens-before and memory accesses, we define a race as follows. Let $A, A' \in \{read, write\} \times OpId$ be memory accesses to some logical location m in an execution. A race exists between A and A' if:

- $op(A) \neq op(A')$ (accesses performed by different operations).
- $op(A) \not\prec op(A')$ and $op(A') \not\prec op(A)$ (the operations are not in the happens-before).
- $kind(A) = write$ or $kind(A') = write$ (one of the accesses is a write).

In the above, op gives the operation identifier for an access, and $kind$ gives the access type (read or write).

Algorithm Next, we define our race detector at the declarative level. We maintain two auxiliary maps $LastRead$ and $LastWrite$ for instrumenting read and write accesses respectively:

$$LastRead \in Loc \rightarrow Id$$

$$LastWrite \in Loc \rightarrow Id$$

Here, $Loc = HElem \cup JSVar \cup Eloc$ and $Id = \{\bot\} \cup OpId$. We use $\bot$ to denote a specially designated value used for initialization. For each location, the map maintains two fields: the first field is the identifier of the operation that last read the location, and the second is the identifier of the last operation that wrote the location. For convenience we define a function CHC to mean Can-Happen-Concurrently as:

$$CHC \in OpId \times OpId \rightarrow Bool$$

$$CHC(A, B) = A \neq \bot \wedge B \neq \bot \wedge A \not\prec B \wedge B \not\prec A$$

Intuitively, two operations A and B can happen concurrently when both are not equal to $\bot$ and (A, B) and (B, A) are not in the happens-before relation.

Our race detection algorithm works as follows: at the start, all entries in both maps are initialized: $\forall e \in Loc$, $LastRead[e] := \bot$ and $LastWrite[e] := \bot$.

Then, upon an access A to an element $e \in Loc$:

- If $kind(A) = read$:
  1. Report a race if $CHC(LastWrite[e], op(A)) = true$.
  2. $LastRead[e] := op(A)$.
- If $kind(A) = write$:
  1. Report a race if:
     $CHC(LastWrite[e], op(A)) = true$, or
     $CHC(LastRead[e], op(A)) = true$
  2. $LastWrite[e] := op(A)$.

That is, on a read, we check whether the last write (if a write occurred) can happen concurrently with the read. If this is the case, we report a read-write race. Similarly, on a write, we check whether the last read (if a read occurred) or the last write (if a write occurred) can happen concurrently with the current write, and if so, we report a read-write or a write-write race respectively.

Note that our race detector only keeps a constant amount of auxiliary information per memory location: a read or a write access will always overwrite its corresponding slot in the location. One advantage of this is that the algorithm will scale well with the number of operations.

Limitation A limitation of our race detector is that it may sometimes miss races. Consider the following example with three operations each performing a single access to location e (for convenience the operation identifier is shown next to the memory access):

$$1\colon\ read\ e \quad || \quad 2\colon\ write\ e \quad || \quad 3\colon\ read\ e$$

Assume that $1 \prec 2$, but that otherwise the operations are unrelated by $\prec$. If the following sequence of operations occurs: $3 \cdot 1 \cdot 2$, the algorithm will not report the race between 2 and 3, since when 2 executes, the detector only has information about the most recent read 1 of e. We plan to address this limitation in future work.

5.2 Implementation

Here, we describe WEBRACER, which includes an implementation of our race detector in WebKit as well as automatic exploration of functionality for simulating user interactions. We note that although we describe a particular race detector, our framework is flexible and allows us to plug in any dynamic race detector (for instance, one could implement an adapted version of FastTrack [7]).

5.2.1 Instrumentation

It was quite difficult to find all relevant instrumentation points in WebKit required to capture the happens-before and memory accesses needed for our race detector. The reason is that WebKit has no single intermediate representation (IR) where all relevant operations are exposed (this is in contrast to say JVM bytecodes, where finding the right bytecodes to be instrumented is straightforward). While WebKit's JavaScript interpreter does have an IR, our detector also requires intercepting HTML parsing, event dispatch, parsing of new JavaScript functions, and so on. Adding instrumentation required careful study of the WebKit code base. We believe that in the future, it would be useful to have a well-defined, standard instrumentation interface for browsers that analysis tools like WEBRACER could be built upon.

Our instrumentation code communicates events directly to the race detector, rather than generating a separate event trace. Internally, the race detector represents the happens-before relation rather directly as a graph structure. While this representation is simple, repeated graph traversals contribute to the high overhead of our implementation (see Section 6); we plan to employ a more efficient vector-clock representation in the future.

5.2.2 Automatic Exploration

WEBRACER allows for manual browsing and interaction with web sites to discover races. To further automate this process, we also implemented *automatic exploration*, which systematically dispatches events corresponding to user actions. Automatic exploration was quite useful for our experiments, as it avoided having to manually trigger each event on each site (a tedious process).

Automatic exploration works by generating any event of certain types for which an event handler was registered by the page. We do all automatic dispatch of events together after the window `load` event is dispatched, simplifying reasoning about WEBRACER's output (since all automatically-dispatched events are together). The events we dispatched automatically were: `mouseover`, `mousemove`, `mouseout`, `mouseup`, `mousedown`, `keydown`, `keyup`, `keypress`, `change`, `input`,

`focus` and `blur`. Additionally, we also generated clicks on links which had JavaScript as protocol in their `href`'s.

We also augmented automatic exploration to expose races on the contents of text boxes and input fields (like the race in Fig. 2). Exposing such races was non-trivial, since simply typing in a text box does not modify the `value` property of the corresponding DOM object (the location that can be accessed by scripts). We solved this problem by adding a handler for the `input` event to all text boxes and input fields which effectively contained the code `this.value := this.value`. With this handler, any typing in a text box immediately updates the corresponding DOM node's `value` property. Finally, we added code to simulate typing into all text boxes.

Overall For instrumentation purposes, we changed roughly 30 `.cpp` and `.h` files in the WebKit source. The changes were not very intrusive – we were able to fairly easily port WEBRACER across several WebKit versions.[10] The total source code for WEBRACER after all modifications, including the race detector was around 2000 lines of C++ code.

5.3 Filters

We also implemented a system for easily adding post-processing filters to WEBRACER's output, to heuristically filter out certain races. Such filters can be useful, e.g., for helping to focus attention on races more likely to reflect application bugs. In running WEBRACER on production web sites written by others, we found the following two filters to be particularly useful:

Focus on form races This filter suppressed all variable races that did *not* involve the value of some HTML form field, e.g., the value in an `<input>` text box. Races on form field values have a high potential to be harmful, as they involve potential side effects of user inputs (see Fig. 2). As an additional enhancement, we further filtered out races on HTML form fields in which the operation(s) writing the form field value v had a read of v preceding the write. Such reads often check to ensure that the user has not modified the field, which makes the race harmless.

Focus on single-dispatch events This filter retained only event dispatch races involving events that dispatch at most once, e.g., the `load` event for a window. Races on such events are more likely to be harmful since once the event is dispatched, an added handler will never be run. In contrast, a `click` event on a button may dispatch multiple times, and missing one of those clicks is less likely to be a serious issue.

Note that we found these filters to be useful specifically for finding harmful races in deployed web sites (see Section 6.3). In a scenario where a developer or tester is checking her own web site for races, alternative filtering may be more suitable.

6. Evaluation

Here we describe an experimental evaluation of our prototype race detector, WEBRACER. In our evaluation, we wanted to test the hypothesis that WEBRACER could be used to find bugs in real web sites, without overwhelming the user with benign race reports. We also wanted to understand better what techniques web developers use to manage asynchronous operations in practice. After describing our experimental configuration (Section 6.1), we present both raw WEBRACER results (Section 6.2) and results with the filtering of Section 5.3 enabled (Section 6.3).

[10] The evaluated version of WEBRACER was built atop WebKit SVN 91698.

Race type	Mean	Median	Max
HTML	2.2	0.0	112
Function	0.4	0.0	6
Variable	22.4	5.5	269
Event Dispatch	22.3	7.0	198
All	47.3	27.0	278

Table 1. The mean, median, and maximum number of races of each type across our test web sites.

6.1 Experimental Configuration

We ran WEBRACER on a set of 100 web sites, comprised of home pages of Fortune 100 companies,[11] and manually inspected the output to find harmful races (to be defined shortly). We ran WEBRACER with automatic exploration enabled (see Section 5.2) to simulate user interaction with each site. This experiment was quite a challenging test, as we were attempting to find bugs in well-tested, deployed web sites whose code was unfamiliar to us.

A key issue in our evaluation was how to determine if a reported race was *harmful*, i.e., whether it indicated a web site bug. In general, making this determination requires knowing the desired semantics of the web site's code. However, we found that discovering relevant semantics by reading a site's code was often infeasible, due to use of complex JavaScript libraries and obfuscating code-compression techniques (e.g., shortening of variable and function names).[12] Given this difficulty, we conservatively classified races as harmful only when they could lead to behavior that was likely to be undesirable independent of application semantics. We defined an HTML race (see Section 2.3) as harmful if it could lead to an attempted update of a yet-to-be-created DOM node, causing a runtime exception. Similarly, a function race (Section 2.4) was harmful if it could cause an invocation of a yet-to-be-parsed function. We discuss harmful variable and event dispatch races in Section 6.3.

6.2 Raw Results

Table 1 gives the mean, median, and maximum number of races of each type reported by WEBRACER across the test web sites, without any filtering.[13] The average number of HTML and function races per site was quite low, making manual inspection manageable. (We discuss the results of that inspection in Section 6.3.) While the median number of variable and event dispatch races per site was still low, several sites had a large number of these races, raising the average number of races per site to 47.3.[14]

Via manual inspection, we found that for many variable races on these sites, the corresponding code was difficult to understand, making it hard to determine if the races were harmful. The difficulty in code understanding stemmed primarily from obfuscation and sophisticated use of asynchronous, delayed script loading, often via complex JavaScript libraries like jQuery.[15] We believe that if a developer were using WEBRACER on their own (un-obfuscated) code, inspection of these variable races would be significantly easier.

[11] At `http://www.srl.inf.ethz.ch/webracer`, we have made available the complete list of the sites that we tested.

[12] Such compression techniques are often used to speed up script downloads.

[13] Note that like many other dynamic race detectors, WEBRACER reports at most one race per location in a given run.

[14] The races reported across different runs for the same site had little variance; our numbers are taken from a typical run.

[15] `http://jquery.com/`

Website	HTML		Function		Variable		EventDisp	
Allstate	**6**	**(6)**	**2**	**(0)**	0		0	
AmericanExpress	**41**	**(1)**	0		0		0	
BankOfAmerica	**4**	**(0)**	**1**	**(1)**	0		0	
BestBuy	0		**2**	**(0)**	0		0	
CiscoSystems	0		**1**	**(0)**	0		0	
Citigroup	**3**	**(0)**	**3**	**(2)**	0		**1**	**(0)**
Comcast	0		**6**	**(1)**	0		0	
ConocoPhillips	0		**2**	**(1)**	0		0	
Costco	**3**	**(3)**	0		0		0	
FedEx	**1**	**(0)**	0		0		0	
Ford	**112**	**(0)**	0		0		0	
GeneralDynamics	0		**1**	**(0)**	0		0	
GeneralMotors	0		**1**	**(0)**	0		0	
HartfordFinancial	**1**	**(1)**	0		0		0	
HomeDepot	0		**1**	**(0)**	0		0	
Humana	0		0		0		**13**	**(13)**
IBM	**16**	**(0)**	0		**1**	**(1)**	0	
Intel	0		**3**	**(0)**	0		0	
JPMorganChase	**3**	**(3)**	**5**	**(0)**	0		0	
JohnsonControls	**1**	**(1)**	0		**1**	**(0)**	0	
Kroger	**1**	**(0)**	0		0		0	
LibertyMutual	0		**4**	**(0)**	0		**1**	**(0)**
Lowes	**1**	**(0)**	0		0		0	
Macys	0		0		**1**	**(1)**	0	
MassMutual	**1**	**(0)**	0		0		0	
MerrillLynch	**1**	**(1)**	0		0		0	
MetLife	0		0		0		**35**	**(35)**
MorganStanley	**1**	**(1)**	0		0		0	
Motorola	**1**	**(0)**	0		0		**1**	**(0)**
NewsCorporation	**1**	**(0)**	0		0		0	
Safeway	0		0		**1**	**(1)**	0	
Sunoco	**11**	**(11)**	0		0		0	
Target	**2**	**(2)**	0		**1**	**(1)**	0	
UnitedHealthGroup	0		0		0		**1**	**(0)**
UnitedTechnologies	**2**	**(1)**	0		0		0	
ValeroEnergy	**5**	**(1)**	**4**	**(1)**	**2**	**(0)**	0	
Verizon	0		**1**	**(1)**	0		0	
WalMart	0		0		**1**	**(1)**	0	
Walgreens	0		0		0		**35**	**(35)**
WaltDisney	**1**	**(0)**	0		0		0	
WellsFargo	0		0		0		**4**	**(0)**
Total	**219**	**(32)**	**37**	**(7)**	**8**	**(5)**	**91**	**(83)**

Table 2. Races reported by WEBRACER on the test web sites, after filtering (see Section 6.3). Sites with no races are elided. The number of harmful races is shown in parenthesis.

We also found that many event dispatch races arose due to deliberate delays in script loading, making those races "benign" with respect to the desired semantics. For example, consider a site s that shows a pop-up menu when the user hovers the mouse over an image i. If s delays loading of the script implementing the pop-up menu (to speed loading of the rest of the page), i may be displayed long before the pop-up menu is available. WEBRACER reports such behavior as a race, and it may indeed be annoying to a user to have degraded functionality as the page is loading (particularly over a slow connection). However, given that the developer must make a deliberate decision to delay script loading, we do not classify such races as harmful.

6.3 Results with Filtering

Table 2 shows the number of races reported by WEBRACER after the filters described in Section 5.3 were enabled, with the number of harmful races shown in parentheses. With the filters enabled, the number of variable and event dispatch races were dramatically reduced, making manual inspection more feasible.

Harmful races We discuss the discovered harmful races of each type in turn.

HTML We found 32 races on HTML elements that could lead to a runtime exception due to access of a non-existent DOM node. They were mostly similar in form to the example in Fig. 3. In some cases, the script performing the DOM node access was loaded asynchronously, making the race less obvious.

Function The seven harmful races were similar to the example of Fig. 4. However, the handler invoking a possibly-undefined function was typically attached to mouse hover or click events, rather than load events. Hence, our automatic exploration (which simulated the clicks and mouse events) was key to exposing these races.

Variable We considered a variable race to be harmful if it could cause user input to be erased, as in the example from Fig. 2. We found five such harmful races on HTML search boxes, and in each case verified that user input during page load would be deleted by a script executing later. The low false-positive rate for this race type (with filtering enabled) indicates that WEBRACER is already an effective tool for finding these high-severity bugs.

Event Dispatch We considered an event dispatch race to be harmful if due to the race, a handler attached to an event might *never* be executed, in accordance with the intuition behind the corresponding filter (see Section 5.3). All the harmful races in this category were due to use of the Gomez performance-monitoring script.[16] Gomez attempts to monitor image load time by checking for new images in the DOM every 10ms (via `setInterval`) and attaching an `onload` handler to each image as it is added. The problem is that if an image loads very quickly, Gomez may add its handler after an image's `load` event has fired. Gomez may have some separate code to compensate for this issue, but we could not tell from the obfuscated versions we were able to inspect.

Benign races The primary cause of benign race reports was synchronization between scripts via data dependence. A canonical example was the Ford site, where all the 112 benign HTML races stemmed from a single pattern, similar to the following:

```
function addPopUp() {
  if (document.getElementById("last") != null) {
    // mutate many DOM nodes
  } else {
    setTimeout(addPopUp,250);
  }
}
```

Via `setTimeout`, the code repeatedly checks if the DOM node `last` has been created, and if so, many other DOM nodes are mutated. The HTML for the page is constructed such that if `last` has been created, all the DOM nodes possibly mutated by the script must exist.

We observe that this commonly used pattern (and variants) operationally encodes a form of ordering between operations: an operation can successfully proceed only if some other operation has executed. To aid human and automated reasoning about such code, it would be very useful if the web platform had an explicit ordering construct for expressing such dependencies in a more declarative manner.

In conclusion, WEBRACER is already an effective tool for finding certain types of harmful races. Further, our data show that benign races are mostly due to either deliberate delays in adding page functionality (see Section 6.2) or synchronization via data dependencies, suggesting directions for future tool improvement. We expect that WEBRACER would be even more effective when used as part of the development process.

Performance WEBRACER was not optimized for performance; we focused instead on achieving good coverage of the many browser instrumentation points required (see Section 5.2). WebKit's just-in-time (JIT) JavaScript compiler was disabled in WEB-

[16] `http://www.gomez.com/`

RACER (as only the interpreter was instrumented), causing a significant slowdown even without instrumentation. WEBRACER's performance was sufficient for finding races in the web sites that we looked at, handling pages with tens of thousands of operations in less than a minute in most cases. However, heavy JavaScript usage incurred significant overhead—we observed a roughly 500X slowdown for the SunSpider benchmarks[17] compared to running with the JIT compiler and no instrumentation. In future work, we plan to develop an optimized implementation that employs more efficient data structures in the race detector (see Section 5.2) and that does not require disabling the JIT compiler.

7. Limitations

Although we have added instrumentation for most commonly-used web features in WEBRACER, we have not yet handled all relevant features (a difficult task, as new features are being added to the web platform rapidly). For instance, WEBRACER's instrumentation does not yet add all relevant happens-before edges corresponding to rule 10 from Section 3.3 for AJAX requests. Also, we have not instrumented calls to `clearTimeout` and `clearInterval` calls [2], which may race with the execution of handlers installed via `setTimeout` and `setInterval`. In general, missing happens-before edges may lead to false positives. Also, our instrumentation does not yet detect DOM node mutation due to application of rules from stylesheets.

In some cases, it is unclear whether certain interactions should be classified as a race. For instance, one can "move" elements with the function `appendChild` (i.e. move an element that is already in the document). If an element is accessed (e.g., via `getElementById`) in parallel with being "moved," then WEBRACER will report a race. However, it is plausible that this interaction should not be classified as a race, since the element existed in the document at all times and was only moved.

Finally, WEBRACER may report false-positive event handler races in cases where the firing of certain events is disabled. For instance, say that a button b is inserted into a document with its visibility set to hidden, disabling clicks. Later, operation 1 adds an `onclick` handler for b and makes b visible, and operation 2 represents a user clicking on b. Since our happens-before relation does not consider the visibility of buttons, we have $1 \not\prec 2$ according to our rules, and WEBRACER will report a false-positive race. In general, detecting disabled events is non-trivial (e.g., clicks may in effect be disabled by shrinking a button), and more robust handling is left for future work.

8. Related Work

Several of the concurrency issues we address were identified informally in a position paper by Ide et al. [11], including races from asynchronous data exchange between a rich client and a Web server and from interleaved HTML parsing and event handler execution (all of which we capture). They advocate higher-level abstractions to handle these issues, but they do not implement either such abstractions or any error-checking tool.

Perhaps the closest work to our own is Zheng et al.'s work on statically detecting races caused by uses of asynchronous requests (so called "AJAX" calls) in JavaScript code [25]. Their system acquires an application's JavaScript code by extracting it from the server-side code (currently, PHP is handled). Their static technique is able to detect AJAX races without having to exercise program behaviors, unlike WEBRACER. Also, their system has special handling for *cookie* state that we have not implemented; adding such handling to WEBRACER would be straightforward. WEBRACER detects a significantly broader class of races than their system (it can catch AJAX races since separate handlers will have no happens-before edges between them). Also, since WEBRACER is implemented entirely in a browser, it works for any server-side technology and for deployed web sites; their current approach requires separate work for each server-side technology. In the future, we plan to investigate how WEBRACER could be enhanced by incorporating static analyses like theirs.

In general, precise detection of the same races found by WEBRACER would be very difficult in a purely static approach like that of Zheng et al., due to pervasive DOM usage (including mutation that may add new scripts) and hard-to-analyze JavaScript features (see discussion in Section 1). The most precise static DOM model we know of is that in Jensen et al. [13], which uses a single abstract representative for all DOM nodes with the same tag (e.g., a single representative for all `<div>` nodes). This abstraction would cause many false positives in a race detector.

There have been several other efforts to detect bugs in JavaScript programs (e.g., [8, 12, 14, 21]), but we are not aware of any previous systems aimed at detecting races besides the Zheng et al. work [25] discussed above.

Recently, a number of novel race detectors for Java-like languages have been developed. For example, FastTrack [7] is a state-of-the art race detector using vector clocks augmented with various optimizations to reduce the space overhead per memory location in the common case to a constant. In our setting, the language is more restrictive as it does not support locks (JavaScript execution is an atomic operation), enabling us to implemented a more specialized race detector. However, in the future, it may be possible to optimize our detector even further.

Our automatic exploration mode (see Section 5.2.2) is similar to the Artemis system [4], which performs feedback-directed execution of event handlers, aiming to maximizing code coverage. Our technique currently does a shallower exploration than Artemis, sufficient for exposing many races. We plan on pursuing deeper automatic exploration techniques like Artemis in the future.

Other previous work has focused on better programming models that enable developers to more easily avoid certain races. Flapjax [18] provides abstractions for reasoning about and composing *event streams* and *behaviors*, allowing for code that, e.g., handles a series of updates from a server without having to manage low-level details of `XMLHttpRequest`s. The Flapjax programming model should help programmers avoid some of the data races that we describe. Mobl [9] takes a somewhat different approach by providing a more synchronous interface to creating Web applications, again abstracting away some of the race-prone features of the web platform. While these systems hold promise for future applications, they do not address the problem of finding races in existing code.

9. Conclusion

The Web platform poses a particular challenge to analyzing concurrency: there are multiple specifications and implementations defining the platform, concurrency is introduced in implicit, event-driven ways, and many different features and APIs are relevant to concurrency. In this environment, we provide the first core tools for reasoning about concurrency: a *happens-before* relation over common HTML and JavaScript constructs, and a *logical model* of memory accesses for capturing state interactions. We show the value of these tools with WEBRACER, the first dynamic race detector for Web applications. WEBRACER found numerous races across top Web sites, including harmful ones capable of causing anomalies like lost input.

Future work includes further automating the detection and possibly remediation of data races in Web applications. Also, at the level of specifications, a precise definition of what concurrent interactions are meant to be allowed would be very useful (we found

[17] `http://www.webkit.org/perf/sunspider/sunspider.html`

defining a reasonable happens-before relation to be surprisingly challenging). Such a definition must provide more clarity to developers, but also not overly restrict the concurrency that has become crucial to performant web applications.

Acknowledgements

We thank Robert O'Callahan for pointing out that concurrency issues can occur in client web applications and for suggesting Firefox test cases as candidates for analysis. We also thank Max Schäfer and the anonymous reviewers for their detailed comments.

References

[1] HTML5 DOM tree. `http://dev.w3.org/html5/spec/Overview.html#dom-trees`.

[2] `setTimeout` specification. `http://www.whatwg.org/specs/web-apps/current-work/multipage/timers.html#dom-windowtimers-settimeout`.

[3] WebKit. `http://www.webkit.org/`.

[4] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A Framework for Automated Testing of JavaScript Web Applications. In *ICSE*, May 2011.

[5] Document Object Model (DOM) Level 3 Events Specification. `http://www.w3.org/TR/DOM-Level-3-Events/`.

[6] ECMA. ECMAScript Language Specification, 5th edition, 2009. ECMA-262.

[7] Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

[8] 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.

[9] Zef Hemel and Eelco Visser. Declaratively programming the mobile web with Mobl. In *OOPSLA*, 2011.

[10] HTML5 specification. `http://www.w3.org/TR/html5/`.

[11] James Ide, Ratislav Bodik, and Doug Kimelman. Concurrency concerns in rich Internet applications. In *Workshop on Exploiting Concurrency Efficiently and Correctly (EC2)*, 2009.

[12] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *ACM Conference on Computer and Communications Security*, pages 270–283, 2010.

[13] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *ESEC/FSE*, 2011.

[14] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural Analysis with Lazy Propagation. In *SAS*, 2010.

[15] Olav Junker Kjaer. Timing and synchronization in JavaScript. `http://dev.opera.com/articles/view/timing-and-synchronization-in-javascript/`. Accessed 03-November-2011.

[16] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[17] Mozilla Developer Network. `https://developer.mozilla.org/`.

[18] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for Ajax applications. In *OOPSLA*, 2009.

[19] Mozilla Developer Network. Avoiding intermittent oranges. `https://developer.mozilla.org/en/QA/Avoiding_intermittent_oranges`. Accessed 18-October-2011.

[20] Robert O'Callahan, December 2010. Personal communication.

[21] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. ADsafety: Type-based verification of JavaScript sandboxing. In *USENIX Security Symposium*, 2011.

[22] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. *SIGPLAN Not.*, 45:1–12, June 2010.

[23] Henri Sivonen. HTML5 script execution changes in Firefox 4. `http://hsivonen.iki.fi/script-execution/`. Accessed 05-November-2011.

[24] Steve Souders. *Even Faster Web Sites: Performance Best Practices for Web Developers*. O'Reilly Media, 2009.

[25] Yunhui Zheng, Tao Bao, and Xiangyu Zhang. Statically locating web application bugs caused by asynchronous calls. In *WWW*, 2011.

A. Happens-Before for Events

Here, we describe how event dispatch works in detail and give some additional (or refined) rules for handling events.

How Events Work Here we briefly sketch event firing; more details can be found in [10, 15]. Firing an event E is done on an element or object T (called a target). Some events can have different phases associated with it: *Capturing*, *At-Target*, *Bubbling* and *Default*. The *Capturing* phase goes through all targets starting at the top of the DOM tree (at the document/window object) and moving down towards the event target T. At each target, all handlers which are registered for that type of event are executed. After the *Capturing* phase, the *At-Target* phase follows, which dispatches all handlers on the event target T. Following is the *Bubbling* phase which works from the target T up towards the document/window object and dispatches the event on all targets on the way. At last, there is the default action which kicks in and dispatches on the target T. For example, on a link element, the `href` may be followed or executed if it is JavaScript code (i.e., of the form `href="javascript:..."`).

To define the happens-before, we first need some definitions. An *inline* event dispatch is the act of programmatically firing an event from JavaScript. An example would be the call `element.click()` in JavaScript code, which fires a `click` event on `element`. Anything else is considered as a *non-inline* event dispatch. An example would be the `mousemove` event firing due to the user moving her mouse, the `load` event for a `window` firing or the `readystatechange` event dispatching on an `XMLHttpRequest` object.

Splitting Happens-Before Since script execution is atomic, the browser will not preempt a script in order to execute a different script, parse more HTML, etc. However, event-handler execution could actually be triggered in the *middle* of JavaScript code due to an inline event dispatch. Let A be such a JavaScript operation that is interleaved with event-handler execution. Given an execution A, we define $A_{[i:j)}$ to mean the subsequence of A starting and including the i'th transition in A and ending with but not including the j'th transition (if $i < 0$, $j \geq |A|$, or $i = j$, then $A_{[i:j)} = \epsilon$). Then, given an execution A interrupted by an inline event dispatch, let the event dispatch invocation be the k'th transition in A's execution. Let B be the set of operations generated by the inline event dispatch. Then: $A_{[0:k)} \prec B$ and $B \prec A_{[k+1:|A|)}$.

When using $A_{[i:j)}$ in the happens-before, we mean that a unique operation identifier is created for this sequence of transitions. Then, in the happens-before as discussed in Section 3.3, we replace an interleaved operation A with the set $\{A_{[0:k)}, A_{[k+1:|A|)}\} \cup B$ (dealing with sets in the happens-before was already explained earlier).

Event Phasing Happens-Before We denote $hop(E, ET, P, T)_i$ as the set of all operations that are the executions of all handlers for the i^{th} dispatch of event E dispatched on target ET in phase P with current target T. Let $A \in hop(E_1, ET_1, P_1, T_1)_i$ and $B \in hop(E_2, ET_2, P_2, T_2)_j$. If E_1 and E_2 are both non-inline events, $ET_1 = ET_2$, $E_1 = E_2$, and $(i < j$ or $(i = j$ and $(P_1 \neq P_2$ or $T_1 \neq T_2)))$, then $A \prec B$.