I Know It When I See It: Observable Races in JavaScript Applications

[Position Paper]

Erdal Mutlu Serdar Tasiran
Koç University
{ermutlu,stasiran}@ku.edu.tr

Benjamin Livshits
Microsoft Research
livshits@microsoft.com

Abstract

Despite JavaScript runtime's lack of conventional threads, the presence of *asynchrony* creates a real potential for concurrency errors. These concerns have lead to investigations of race conditions in the Web context. However, focusing on races does not produce actionable error reports that would at the end of the day appeal to developers and cause them to fix possible underlying problems.

In this paper, we advocate for the notion of *observable races*, focusing on concurrency conditions that lead to visually apparent glitches caused by non-determinism within the runtime scheduler on the network. We propose and investigate ways to find observable races via *systematically exploring* possible network schedules and *shepherding* the scheduler towards correct executions. We propose crowd-sourcing both to spot when different schedules lead to visually broken sites and also to determine under what environment conditions (OS, browser, network speed) these schedules may in fact happen in practice for some fraction of the users.

1. Introduction

Today's client-side Web has developed largely without much concern around multi-threaded execution.

When it comes to client-side Web programming, today JavaScript powers the majority of large and popular Web sites. JavaScript execution is single-threaded. Yet the complex needs of sites such as Facebook, Outlook, Google Maps, and the like have lead to *asynchrony* becoming a common way to program complex Web applications. It is asynchronous processing that is responsible for interactive and responsive user interfaces (UIs) that operate without blocking the UI thread or requiring a reload, as did Web applications of the late 1990s. JavaScript's event-driven execution model matches the asynchronous call paradigm very well. Large JavaScript applications extensively use callbacks: function closures that are placed on the event loop for later processing. Here, a network connection is open and a `onreadystatechange` handler is set up to process return data as it is returned by the network. Perhaps the most common use of asynchrony is JavaScript Web applications communicating with back-end servers using a `XMLHttpRequest` (XHR) object, as shown in Figure 1.

Despite JavaScript lacking conventional threads, the presence of asynchrony creates a potential for races, in a way that is similar to concurrent code running on a single processor. In particular, the ordering of event execution in JavaScript as well as the timing of completions of asynchronous requests is non-deterministic, prone to be affected by network timing, etc. Asynchrony in JavaScript programs, therefore, may result in concurrency-related errors.

Prior work: Zheng et al. [9] propose a static technique to detect potential races in JavaScript applications. More recently, Petrov et al. [6] and Raychev et al. [7] 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 (giant blob of) global state, creating the potential for races.

Petrov et al. [6] 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 [7] identify and filter out large sets of benign races.

Despite these significant efforts, we are not aware of any techniques of this sort being applied in practice. We conjecture that the potential for high false positive rates

```
var req = new XMLHttpRequest();
req.open("GET",
    "login.php?name="+ username, true);
// set up the callback
req.onreadystatechange = function () {
    if(req.readyState == 4) {
        // do something
    } };
req.send(null);
```

Figure 1: Typical use of XHR.

and the need for result post-filtering [6, 7], combined with a lack of prioritization of warnings, does not result in very actionable results. Put one way, the approaches above take a *low precision/high recall* approach to race detection. In contrast, the focus of this work is on a *high precision/low recall* oracle for harmful races, as well as *recovery* techniques.

What is the damage: As is the case for shared-memory concurrent programs, race conditions in Web pages are low-level events that may be symptomatic of higher-level concurrency and design errors. However, race conditions are only a *proxy* for serious concurrency errors and can be inaccurate as a correctness criterion in two ways:

- **False positives:** as investigated in [6, 7] race conditions may be entirely *benign* and not lead to user-observable errors, data corruption or loss. Moreover, Web users generally have lower expectations of robustness than users of systems code or, say, compilers. The most common reaction to a misbehaving Web application may be to just reload the underlying page.
- **False negatives:** fixing a race condition may not fix the real concurrency error. In an example reported in [9], a request issued by the user via a mouse click is applied *not* to the data currently displayed to the user, but to data that is in the process of being received. In this example, the real bug is an atomicity violation and a race condition is the symptom. In examples such as this one, it is possible to fix the race condition (the symptom) but not the actual bug.

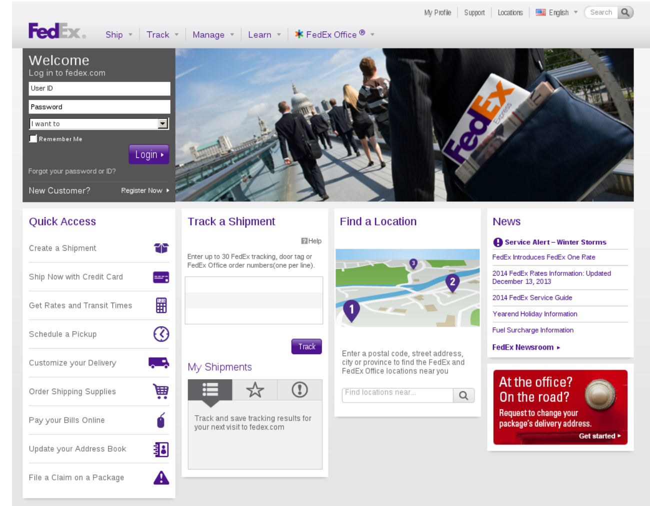
We argue that so far, there has not been a satisfactory oracle proposed that distinguishes benign and harmful Web races proposed in the literature. In this paper, we endeavor to develop such an oracle.

Observable races: Our approach is the mirror opposite of the one in prior work outlined above: instead of finding potential races and then filtering them to focus on the more damaging ones, we *start* with the category of races that is decidedly noticable by the user. We wish to find races that are both provably damaging and fixable; it is not our goal to be exhaustive.

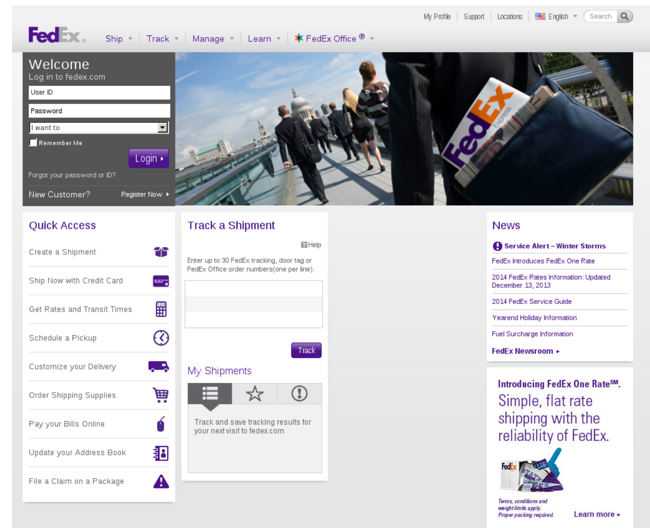
We call races that lead to buggy behaviors that are visible to the end-user *observable races*. A fundamental issue with bug detection tools, both static and runtime, is *explaining* analysis results in a way that is understandable by the developer, and ultimately leads to a fix. Focusing on observable races naturally allows us to create *repros* for our findings, i.e. visually obvious proofs of there being a race[1].

For an example of a behavior discrepancy that is visible to the end user and results in a semantically different output for the web page, consider the two screenshots shown in Figures 2a and 2b. These screenshots were obtained from two different XHR timings (shown in Figure 3a and 3b) on an artificially-broken version of the FedEx web site, in which we replaced some synchronous XHR calls with asynchronous ones in order to be able to illustrate the semantic visual

[1] Admittedly, some "deeper" properties such as "does this race lead to monetary loss" or "does this race lead open up a security vulnerability" cannot be addressed with our definition.



(a) Screenshot for one unmodified XHR call timing.



(b) Screenshot for an alternative, randomly-delayed XHR timing.

Figure 2: Screenshots obtain for two different XHR timings for an artificially-broken version of the FedEx Web site.

differences (the "smoking gun") that we target with our approach.

1.1 Exposing Races with Systematic Exploration

In order to trigger different concurrency-related behaviors of a Web application, we effecively virtualize the network interface exposed to JavaScript applications via XMLHttpRequest (XHR). We provide a mechanism to wrap XHR calls and systematically explore all possible XHR call orders. Our technique uses a proxy based dynamic instrumentation tool [4] to employ source-level instrumentation for controlling XHR calls.

(a) Well-behaved timing.

(b) Defective timing resulting in observable race.

Figure 3: Original and modified XHR timings for the artificially-broken version of the FedEx site.

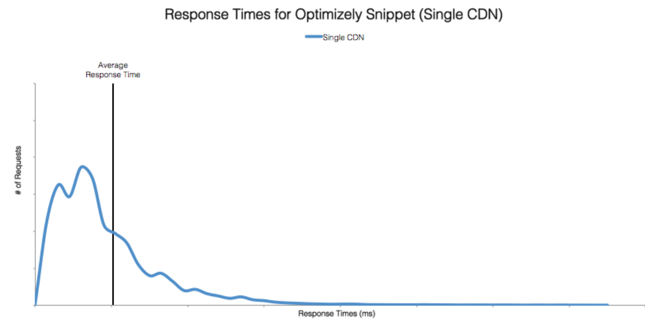


Figure 4: Overall distribution of response times for the Optimizely snippet worldwide, with the average response called out explicitly from `blog.optimizely.com/2013/12/11/why-cdn-balancing/`.

In our initial effort (Section 2), we investigated different orderings of XHR calls using a randomized exploration mechanism by applying randomized delays to each XHR call. We experimented with XHR-heavy Web sites chosen from the Alexa index and we explored a large number of XHR orderings in a brute-force manner. Surprisingly, even on sites in which this approach was able to explore all possible XHR interleavings, we did not detect any *observable races*. While this might be possibly be explained by the fact that for Web sites with observable races, the brute-force approach did not exercise any of the observable race scenarios, we believe this explanation to be unlikely. We instead conjecture that observable Web races in the wild are extremely rare. In this work, we build a systematic XHR schedule exploration tool (Section 3) that, combined with the rest of our approach described below, will serve as a detection tool for observable races. We are going to use this tool to test our conjecture that observable races are extremely rare.

Exhaustive systematic exploration of XHR schedules concretely is not the only way to detect observable races; indeed, one could perform symbolic execution or multi-execution, both of which have been tried for JavaScript before [5, 8]. Once we have at least two different XHR timings on a given application for which there is indication that the visual outputs are different, we can compare the outputs in a more semantic manner.

I know it when I see it: The most direct way to perform such a comparison is to capture *screenshots* that the different XHR call orders result in, in an effort to spot visual differences. There are several approaches to being able to spot visual differences due to races:

- the developer can be involved in the process, yet the number of schedules may prove so large that considering all before/after screenshots will be prohibitively costly;
- screenshot differencing may be done automatically [1]. While in many cases the before and after images are exactly the same, in some cases there are subtle differences. Moreover, these differences are often semantic: a slight difference in page layout such as a table border being moved by a millimeter may result in a lot of pixels being different between the two images;
- the task of screenshot differencing can be given to humans who are, we argue, better suited to the task of determining which differences are semantic and which are minor differences in appearance.

Our current implementation uses a combination of the last two approaches to build a semantic oracle: if the before/after images are not the same, we ask crowd workers to decide if they are indeed semantically different and, if so, how. Note that the issue of network non-determinism needs to be addressed: in general, we need to replay the same network responses if we hope to get the same visual result. This is because the majority of popular Web pages use ads, which often change on every page load and need to be memoized and replayed back.

1.2 Smoking Gun

The *repros* described above usually take the form of a before/after screenshot pair, the before looking as expected by the user, the after being broken in some way. They play a tremendous role in creating convincing and actionable error reports.

It is important to recognize that there is a fair bit of inertia when it comes to trusting the results of analysis tools, especially static analysis tools that reason about "the hypothetical." This is due to an informal belief that the bug will not occur in practice. But more fundamentally, even in safety-critical code, developers are often fearful of fixing races, unless they can be be clearly convinced of the damage, for fear of introducing new errors, unmasking other errors, introducing performance or security issues, etc.

It is our position that showing screenshots with visual defects to the developer would serve as a more convincing argument than presenting hypothetical races found via either static analysis or runtime exploration. It is especially important to note that developers might be reluctant to consider results they consider hypothetical for applications that have already been deployed and tested in the field. To sum-

marize, our ultimate goal is to present the developer with a combination of the following as an error report:

- Scenario description (workload and at least two different XHR schedules);
- implicit constraints being violated, such as a happens-before relation that does not hold for the offending execution;
- before/after screenshots;
- environment/network conditions that lead to the screenshots above.

1.3 Prioritizing Races Using a Crowd

Finding an error "in the wild" as opposed to finding one in an execution that has never been observed in real-world conditions will lead to bug reports of significantly higher intrinsic value. We can use a crowd of users not only for distinguishing between "good" and "bad" screenshots of a Web application shown in browser. We can, in fact, use a crowd for *confirming* a particular race as observed in an actual execution. A key element of this strategy is to record (and profile) crowd workers based upon their environment characteristics such as the OS they use, the browser they run, the device from which they access the Web, as well as network characteristics (3G, 4G, WiFi, T3, T1).

One way of using the crowd for exploring the landscape of interleavings possible in the wild is to

- wrap XHR calls with instrumentation and logging code, and have the crowd members log the order of callbacks they experience,
- ask users or have a runtime monitor whether a harmful outcome was observed, and
- store both buggy and well-behaved interleavings, along with the user's configuration information.

This can be seen as crowd-sourcing the exploration of different schedules induced not by systematic exploration but by different configurations, in addition to crowd-sourcing the determination of whether a Web page output is semantically broken or not.

The challenge of this strategy will be in exploring the long tail of the population when it comes to these characteristics. In particular, it is easy to explore the common case such as Internet Explorer 10, 32-bit on Windows 7 or Chrome 22 on MacOS X, but finding someone who uses a particularly uncommon version of a rare browser on a slow connection is likely to be both more challenging and more fruitful, as these configurations are unlikely to have surfaced during testing. Being able to selectively solicit users with uncommon evironment combination is key in making this approach work well.

If we also have statistics about what percentage of users are in every segment (OS, browser, device, network speed), we can also extrapolate to predict how common every race may be in practice based on the data we sample via the crowd. Developers would find this information useful for prioritizing bugs.

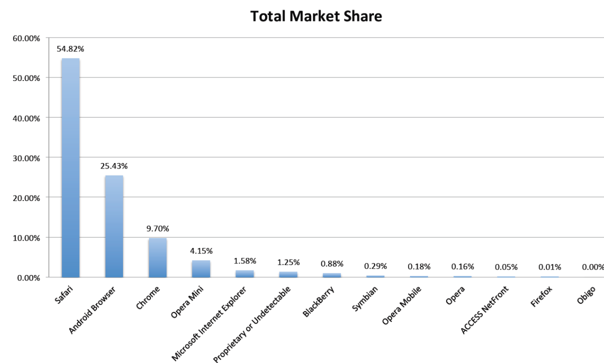


Figure 5: Total mobile browser distribution.

1.4 Schedule Shepherding

For Web applications in wide use, it is expected that in "most common cases" the Web page must not be resulting in serious concurrency errors. We build upon this intuition and record "well-behaved" schedules/interleavings for a Web application, i.e., interleavings that do not lead to outcomes identified as errors during crowd-sourcing. We propose the use of *schedule shepherding* to avoid observable races.

Based on our brute-force randomized exploration experiments, we conjecture that *observable races* are rare events. Yet, they may occur for widely-deployed applications because of the inherent *diversity* of devices, browsers, operating systems, and network conditions. While races in deployed applications are unlikely to appear for the most common configurations, we observe a long-tail phenomenon in terms of response latency (Figure 4) and also the browser being used (Figure 5). As a result, uncommon configurations may not be covered in testing, leading to the potential for rare, but damaging races in deployed code. Moreover, even if races happen in the field, perhaps, for uncommon configurations, users are likely to chalk whatever they see up to "random glitches" so common in today's Web and not report them to the developer. It is our goal to shepherd the uncommon but buggy interleavings towards well-behaved ones.

By orchestrating the order of execution of wrapped XHR calls (by delaying the execution of a callback, if necessary) we can *guide* or shepherd the execution order of XHR responses/callbacks toward known, well-behaved schedules. While such an approach would not be viable for extremely critical systems software, it provides a lightweight fix for concurrency errors that might otherwise remain completely unaddressed.

1.5 Key ideas

To summarize, the key ideas put forth in this paper are

Observable races. The notion of observable races, as determined by visual inspection of the Web application.

Systematical exploration for XHR call orders. A mechanism for systematically exploring all possible orderings of asynchronous calls in order to exercise potentially defective schedules.

Crowd-sourcing for finding concurrency bugs. A crowd-sourcing approach for determining whether a concurrency error has resulted in observable non-determinism.

Smoking gun detection We propose ways to find observable data races "in the wild", as opposed to in hypothetical, synthetic executions. This approach involves use a crowd of users in a different capacity than before.

Schedule shepherding. Shepherding of Web-page non-determinism in order to avoid outcomes determined to be harmful via crowd-sourcing.

2. Randomized Exploration

For our randomized exploration, we focused on Web sites from different contexts. We analyzed some Web sites that have been reported to contain data races that are classified as harmful in prior work [6, 7]. We also investigated Web sites that make extensive use of XHR calls (Figure 6) that we obtained by analyzing the Alexa top 4,000 sites. With the help of scriptable WebKit mechanisms [2, 3], we were able to inject our wrapped XHR call methods for applying our randomized delay strategy to Web sites under test and also create Web site screenshots at the end of the page load event.

We first provide key features of our approach for determining observable races. In order to identify widely used, XHR-call-heavy Web sites, we examined the Alexa top 4,000 site list. Out of 4,000 Web sites, we selected 900 Web sites containing at least one XHR call. We believe we would have ended up with a higher count if we had explored sites that require authentication such as Outlook, Facebook, etc. We used an automated network traffic recording mechanism that collects HTTP Archive (HAR) file for each Web site in the Alexa top 4,000 sites. After analyzing the collected HAR files, we were able to count the number of XHR calls made for each Web site. For illustration, Figure **??**

Figure 6 presents data about sites we investigated that are heavily-dependent on XHR calls. The table provides information about the number of times XHR was called at runtime as a result of visiting the page. Additionally, the table shows statistics about the mime types of the responses obtained via XHR and also the types of the XHR calls made (XHR calls can also be made synchronously). This allows us to distinguish between JSON, JavaScript code, and plain text being returned. In order to be able to trigger different response time orderings, we wrapped XHR calls with randomized delays. To record the visual consequences of different XHR orderings, we captured the screenshot of the loaded Web site with and without randomized delays. We automated this procedure for easy repeatability. Our automated procedure works as follows;

1. We capture network traffic records and two screenshots of a Web site by loading it twice before applying any delay mechanisms
2. We capture the network traffic record and the screenshot of the Website after our randomized delays applied.
3. We eliminate cases in which the unmodified and randomly-delayed versions of the page produce the same screenshot by using a masking mechanism [1].

```
req.onreadystatechange = function () {
        if(req.readyState == 4) {
            // first handler
            var req2 = new XMLHttpRequest();
            req2.onreadystatechange = function () {
                // second handler
                ...
            }
        }
    };
}
```

Figure 7: XHR chaining.

4. We then crowd-source the determination of whether the randomly-delayed version of the Web page is broken.

Although, in theory, there are $n!$ possible XHR call orders for n asynchronous XHR calls, in practice this number can be lower as Web application developers use *chained* XHR calls where one callback function initiates another XHR call. Chaining creates constraints that restrict the number of possible schedules. An example of chaining is shown in Figure 7. The order of `onreadystatechange` handler execution is preserved.

For our randomized exploration experiment, we selected a couple of Web sites with an average number of XHR calls. A representative example is `http://www.news24.com` with 7 asynchronous calls which should have at most $7! = 5,040$ possible orders. We and applied randomized exploration mechanism without any prior analysis for *chained* XHR calls. Even though we employed our randomized mechanism in a brute-force manner on these sites, we were only able to investigate around $1,600/5,040$ distinct XHR call orders without observing any visual defects on the generated screehshots.

Thus, we conjecture that *observable races* are rare events and to uncover such events systematical exploration with the knowledge of *chained* XHR calls is needed.

3. Systematic Exploration

For our systematic exploration, we carried out a source-level instrumentation of XHR calls using a proxy-based dynamic instrumentation tool for JavaScript, *AjaxScope* [4]. This approach uses a client-side proxy positioned between the Web server the browser to dynamically capture and rewrite JavaScript code. We wrapped XHR calls with additional recording mechanisms for capturing the XHR scheduling order of a viewed Web page.

Our exploration starts by recording an initial order of the XHR calls observed during the loading of the page using AjaxScope instrumentation. We also devised a logging mechanism for capturing information about *chained* XHR calls. When systematically exploring XHR schedules, these chaining dependencies are taken into account so our exploration tool never attempts to exercise two chained XHR calls in the wrong order.

Starting from the recorded initial XHR order and *chained* XHR call information, our exploration will generate all possible XHR schedule permutations to be tested. For each gen-

Web Site	Response mime types			Classifying XHR		
	JSON	JavaScript	Text	Sync XHR	Async XHR	Total XHR
*.mlb.com	9	2	24	10	25	35
discussions.apple.com	26	0	0	0	26	26
www.aljazeera.net	17	1	5	0	23	23
www.gazzetta.it	3	0	19	10	25	22
wireless.att.com	3	0	15	2	16	18
www.welt.de	0	0	18	0	18	18
www.tvguide.com	1	5	11	0	17	17
www.optimum.net	3	1	12	0	16	16
www.fujitv.co.jp	7	8	1	3	13	16
www.bild.de	2	0	12	0	14	14
www.nasa.gov	12	0	2	0	14	14
news.qq.com	0	0	12	0	12	12
www.zaobao.com	0	0	12	0	12	12
www.girlsgogames.com	9	0	3	4	8	12
www.sports.ru	3	0	8	0	11	11
www.premierleague.com	2	0	9	2	9	11
www.eltiempo.com	0	0	10	0	10	10
www.myvideo.de	9	0	0	0	9	9
www.politico.com	0	1	8	0	9	9
www.att.com	3	0	6	2	7	9

Figure 6: Top 20 XHR-heavy Web sites in our experiments.

erated schedule, our tool enforces the corresponding schedule and takes a screenshot of the resulting Web page on the browser. In order to load a Web page with a desired XHR call schedule, we instrumented each XHR callback function to be executed as according to the provided order. Given a schedule, our instrumentation controls the execution of each triggered callback by cross-checking with the schedule. We give each XHR a unique ID. Consider a given desired order of execution of XHR callbacks, expressed as a permutation of XHR IDs. When responses from the network arrive, the corresponding callbacks are not immediately executed. Instead, the execution of the callback for XHR with ID i is delayed until i is the next ID in the given permutation.

Below is the step-by-step explanation of our mechanism:

1. Capture initial order of XHR calls and the information about *chained* XHR calls using instrumentation provided by AjaxScope.
2. Depending on the initial order and information about *chained* XHR calls, generate all possible XHR schedules for exploration.
3. For each generated schedule, enforce that Web page is loaded with XHRs following this schedule, delaying them as needed.
4. Collect screenshots for each enforced schedule to be checked for *observable races*.

4. Discussion

Compared to other runtime environments, concurrency in the browser has not been exhaustively specified or formalized. When it comes to the issue of *scheduling*, nondeterminism can be caused by both the built-in scheduler and environment conditions such as the order of network message arrival. While researchers have explored the issue of runtime races, their findings have not proven actionable for the Web developer for two reasons. First, because of a high false positive rate, the developer does know know which races to fix. Second, due to the absence of explicit concurrency primitives for programming on the Web platform, the developer does not know how to fix them. While it is tempting to ask for tools and solutions, we claim that the sources and extent of the problem are far from being well understood at this point.

Our long-term research involves answering the following research questions.

Schedule diversity landscape. What kind of schedules are common in complex Web applications? Is it the case that most or all widely-used browsers effectively follow the same schedule or are there significant differences in terms of orders that result from Chrome vs. Firefox? What is the density distribution like? Is it the case that there is a *long tail* in terms of observed schedules?

Smoking gun. Do obscure, synthetically-generated schedules actually correspond to schedules that may be observed *in the wild*? Can we actually detect these uncommon races in the wild, recoding the conditions under which they are possible?

Finding a crowd. Can we "design" a user population that would allow us to experimentally confirm such races? For example, perhaps, choosing users of Chrome on the iPad will produce unusual schedules, so deliberately targeting such users may be fruitful.

Observability vs. damage. Can we correlate races to *damage* to the user, application state, etc. It is not obvious what the "worst thing that can happen" is. Is it just a broken page or can observable races lead to logical bugs resulting in, say, financial losses or, possbily, enable security vulnerabilities?

Better runtime monitors. While our focus has been on *visually* observable races, we can also explore runtime monitoring as an oracle for finding damaging descepancies due to scheduling differences.

5. Conclusions

This paper presents both our short-term research exploration and long-term vision for understanding and possibly fixing concurrency errors in Web applications.

References

[1] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Flowfox: A web browser with flexible and precise information flow control. In *Proceedings of the ACM Conference on Computer and Communications Security*, CCS '12, pages 748–759, New York, NY, USA, 2012. ACM.

[2] A. Hidayet. Phantomjs. `http://phantomjs.org/quick-start.html`.

[3] L. Jouanneau. Slimerjs. `http://docs.slimerjs.org/0.9/release-notes.html`.

[4] E. Kiciman and B. Livshits. AjaxScope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 17–30, New York, NY, USA, 2007. ACM.

[5] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Proceedings of the IEEE Symposium on Security and Privacy*, SP '12, pages 443–457, Washington, DC, USA, 2012. IEEE Computer Society.

[6] 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*, PLDI '12, pages 251–262, New York, NY, USA, 2012. ACM.

[7] 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 & Applications*, OOPSLA '13, pages 151–166, New York, NY, USA, 2013. ACM.

[8] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.

[9] 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*, WWW '11, pages 805–814, New York, NY, USA, 2011. ACM.