NodeRacer: Event Race Detection for Node.js Applications

André Takeshi Endo*†, Anders Møller*
*Aarhus University, Denmark
†Federal University of Technology – Paraná, Brazil
andreendo@utfpr.edu.br, amoeller@cs.au.dk

***Abstract*—The Node.js platform empowers a huge number of software systems programmed with JavaScript. Node.js employs an asynchronous execution model where event handlers are scheduled nondeterministically, and unexpected races between event handlers often cause malfunctions. Existing techniques for detecting such event races require complex modifications of the Node.js internals, or target only certain kinds of races.**

This paper presents a new approach, called NODERACER, that detects event races in Node.js applications by selectively postponing events, guided by happens-before relations. The technique is implemented entirely with code instrumentation, without modifications of the Node.js system. Our experimental results give evidence that NODERACER finds event race errors with higher probability than a state-of-the-art fuzzer, and that the use of happens-before relations helps avoiding false positives. Furthermore, we demonstrate that NODERACER produces actionable error reports, and that it can be helpful for detecting test flakiness that is caused by event races.

***Index Terms*—JavaScript, race conditions, flaky tests**

I. Introduction

JavaScript is a multi-paradigm dynamic programming language that was initially designed to program client-side web applications. The launch of the Node.js platform in 2009 helped spread Javascript to other domains, including server-side applications, command line tools, and even desktop apps. Node.js along with its official package manager npm[1] is one of the largest and most widely used software ecosystems according to Stack Overflow[2] and GitHub,[3] with more than 1 million npm packages now available and more than 10 billion package downloads weekly.[4]

It is well-known that the JavaScript execution model is susceptible to races [1]–[8]. In JavaScript, race conditions arise not because of multiple threads accessing shared memory concurrently, but because the program code is executed asynchronously. In general, any task that may take some time to be processed is handled as callbacks that respond to events. For example, to access data from the network, an event handler callback can be registered, such that when the network data is ready, the event handler is executed to process the data. Event handlers always run until completion, non-preemptively. When multiple event handlers are registered, the execution order among the different event handlers—and thereby the behavior of the application—can depend critically on the exact timing of the events. Interleavings that were unexpected by the programmer may take the application to a faulty state.

Event race errors are often difficult to detect or reproduce during ordinary testing at development time, because such testing is usually performed in a controlled environment, with a fast and predictable network, reliable databases, etc. Event race errors in client-side JavaScript code may result in crashes and annoyed users, but simply reloading the web page and repeating the user interactions is often sufficient to recover. In server-side JavaScript code, the stakes are higher: event race errors do not just affect individual users but may crash the entire system, or even worse, lead to security vulnerabilities [9]. Event races are also a well-known cause of flaky tests: nondeterministic behavior of regression tests often arises due to races between the test code and the system under test [10], [11]. For these many reasons, it is important to provide means for Node.js developers to better detect event races and diagnose their causes.

Race detection for JavaScript has been mostly investigated for client-side code in web applications. Predictive techniques are based on the idea that, based on a sample run of the web application, it is possible to record and analyze the memory accesses and the causal dependencies between the event handler executions, and from that information predict race issues in other runs [2], [3], [12]. It has been observed that such techniques often result in too many false positives, and that it is often difficult to diagnose the causes of the reported races and thereby to fix the errors [5], [7]. For these reasons, other techniques are designed to explore different interleavings and produce witnesses of likely harmful behavior, such as uncaught exceptions or inconsistent GUIs [4]–[7], [13]. Although the asynchronous execution models of browsers and Node.js are conceptually very similar, it is nontrivial to adopt those techniques to Node.js.

Despite the massive prevalence of Node.js, only a few race detection techniques have been developed for this platform. NodeAV [14] focuses on atomicity violations and does not consider other kinds of races. Node.fz [15] detects races by incorporating fuzzing into the Node.js scheduler. By randomly shuffling the entries in the event queues, it increases the

This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 647544).

[1]https://www.npmjs.com/
[2]https://insights.stackoverflow.com/survey/2019
[3]https://octoverse.github.com/
[4]https://snyk.io/blog/npm-passes-the-1-millionth-package-milestone-what-can-we-learn/

probability of encountering a failing execution during testing. This elegant approach can reveal many race bugs, however, by only shuffling event handlers that are ready to execute, it misses bugs that only manifest when specific events are delayed sufficiently, which we demonstrate in Section II. Also, due to the aggressive fuzzing, it can be difficult to faithfully reproduce failing executions and to locate the root causes of the failures being provoked. Furthermore, despite the conceptual simplicity of the approach, the implementation of Node.fz requires 10 000 lines of code because of the complex internals of Node.js and the need for avoiding event schedules that are impossible in ordinary executions.

In this paper, we introduce NodeRacer, a new approach to detect event races in Node.js applications. The key idea is to explore alternative schedules entirely by selectively postponing events, without interfering with the internal event queues in Node.js. This approach is inspired by EventRaceCommander [8] and AjaxRacer [6]. To avoid postponing events in ways that are impossible in ordinary executions, and to avoid postponing events needlessly, the exploration is guided by a happens-before relation that captures causal relationships of events. The happens-before relation is inferred from an initial execution of the application, driven either by a regression test or by a manual interaction with the application. Unlike traditional race detectors [2], we follow the approach of AjaxRacer [6] by completely ignoring the low-level data races on individual memory locations and instead detecting race errors by looking for the observable effects, such as crashes or regression test failures. To support debugging after a race error has been detected, NodeRacer can be run in a diagnosis mode where it systematically looks for the smallest number of changes to the original schedule that exposes the error, inspired by the conflict-reversal bounding technique of R^4 [7]. The approach is implemented entirely in JavaScript, without changes to the Node.js internals.

In summary, the paper makes the following contributions.

- We propose a light-weight event race detection technique designed for Node.js applications that is based on the idea of selectively postponing events to explore executions that are difficult to reach in ordinary testing.
- We show that happens-before relations between events can be inferred using low-overhead instrumentation and used for guiding which events to postpone. As part of this, we present happens-before rules for Node.js that extend and improve those from previous work.
- We present the results obtained using our prototype implementation, demonstrating that (1) NodeRacer finds event race errors with higher probability than Node.fz, (2) the use of happens-before relations helps avoiding false alarms, (3) the error reports provided by NodeRacer are useful for diagnosing the causes of event races, and (4) NodeRacer can be useful for detecting test flakiness that is caused by event races. The implementation and all experimental data are openly available.

Code from the Archiver library

```
 1 var doneTasks = 0;
 2 var entries = 0;
 3
 4 function processFile(filePath) {
 5    entries++;
 6    fs.lstat(filePath, function stat(err, stats) {
 7       if (err) {
 8          entries--;
 9          return;
10       }
11       useStatData(stats);
12       fs.readFile(filePath, function read(err, data) {
13          performTask(data);
14          doneTasks++;
15          if (doneTasks === entries)
16             finalize();
17       });
18    });
19 }
```

Code from one of Archiver's test cases

```
20 processFile('existing-file.txt');
21 processFile('missing-file.txt');
22 processFile('empty-file.txt');
```

Fig. 1. Motivating example.

II. Motivating Example

The JavaScript code in Figure 1 illustrates a typical Node.js event race error. This particular example is based on a previously unknown bug in the Archiver[5] library that was uncovered by NodeRacer. The first part of the code shows the relevant parts of the library, and the second part represents the test NodeRacer used to reveal the bug.

The function `processFile` receives as argument the path to a file (line 4), increments `entries` (line 5), and invokes `fs.lstat` (line 6) from the Node.js API to obtain stats information on the file. The callback `stat` (lines 6–18) is run when the stats on the file are collected: if there is an error, it decrements the entries and returns (lines 7–10); otherwise, it processes the stats (line 11) and invokes `fs.readFile` (line 12) from the Node.js API to read the contents of the file. When the file contents are available, callback `read` (lines 12–17) is executed: it performs some task with the file contents (line 13), increments `doneTasks` (line 14), and calls function `finalize` (line 16) if the number of tasks done is equal to the number of entries (line 15). The operations `fs.lstat` and `fs.readFile` are asynchronous, so their callbacks are executed when their responses are ready.

The test code (lines 20–22) calls `processFile` in sequence three times for an existing file, a missing file, and an empty file, respectively. By running it, five callbacks are produced: three instances of `stat`, we refer to them as $eStat_{ex}$, $eStat_{ms}$, and $eStat_{ep}$ for existing file, missing file, and empty file, respectively, and two instances of `read`, we refer to them as $eRead_{ex}$ and $eRead_{ep}$. There is no invocation of `read` for the missing file since $eStat_{ms}$ is called with an error and no callback is registered in that case. These five callbacks may interleave with each other, the only two restrictions are

[5]https://github.com/archiverjs/node-archiver

that $eStat_{ex}$ registers (happens-before) $eRead_{ex}$ and $eStat_{ep}$ registers (happens-before) $eRead_{ep}$.

The race bug manifests when $eStat_{ms}$ is the last callback executed; it correctly updates the number of entries (line 8) but fails to call the `finalize` function. In the actual Archiver code, this causes the system to hang.

This bug may not be detected by existing race detection techniques for Node.js. Variables `doneTasks` and `entries` represent common cases of control and synchronization mechanisms between callbacks. Such cases hinder the application of predictive approaches: several callbacks are conflicting because they operate on the same variables, and it is difficult to reason about which conflicting callbacks are actually harmful.

Also, fuzzing the Node.js scheduler as done by Node.fz may still not be enough. For this example, a typical run will have the `stat` callbacks ready to execute at almost the same time, and then after some time the `read` callbacks are ready. This means that the shuffling of the event queues will concentrate on those two subsets of callbacks, failing to mix and shuffle all of them. As explained above, this bug is revealed only if $eStat_{ms}$ is sufficiently delayed to be scheduled last. For this reason, Node.fz does not find the bug even after $1\,000$ runs.[6]

III. Background

Node.js is a JavaScript-based runtime environment that integrates three main components [16]: *(i)* the V8 high-performance JavaScript engine developed by Google; *(ii)* the *libuv* I/O library that implements the event loop, the worker thread mechanism, and all of the asynchronous behavior; and *(iii)* native libraries that abstract the functionalities for high-level use by developers. On top of this, the npm repository provides a huge collection of open source libraries.

Node.js adopts an event-driven architecture where callbacks (also known as event handlers) are registered to execute when relevant events occur [17]. In this paper, a callback is the runtime instance of a JavaScript function call along with its execution context. Callbacks are scheduled to run by a single-threaded event loop; when a callback is picked, Node.js runs it to completion without interruption in one tick of the event loop. Node.js has three main categories of asynchronous execution functionality that are sources of nondeterminism and event race errors: I/O operations and related computational resources, timer and process-related operations, and promises. The happens-before rules defined in Section IV-B involve all three categories.

I/O and related operations are performed by a pool of worker threads that run without blocking the main event loop. As these operations are executed concurrently by the workers and depend on external computation, there are generally no guarantees about the execution order of the callbacks.

[6]Node.fz sometimes delays timer events for 5 ms, and it similarly injects a 0.1 ms delay in the worker thread before each task is processed, when using the default configuration. Those delays are not always the right ones needed to reveal the race errors.

The `timer` module and the `process` object provide functionality for scheduling callbacks to be called at some future period of time [18]. The main functions are:

- `setImmediate(`cb`)` registers callback cb of type Immediate to be run once after the I/O-related callbacks that are ready to execute.
- `setTimeout(`cb`,` d`)` registers cb of type Timeout to be run once after a delay of (at least) d milliseconds.
- `setInterval(`cb`,` d`)` is similar to `setTimeout` but repeats indefinitely.
- `nextTick(`cb`)` registers cb of type TickObject to be called once the current tick of the event loop is completed. It differs from the other functions by not allowing other callbacks to be scheduled between the current tick and the one that runs cb.

Promises are JavaScript objects that represent the eventual completion or failure of an asynchronous operation, which are useful for structuring asynchronous computations [19]. When a promise is created, callbacks can be registered to be run after successful (fullfilled) and failing (rejected) executions, using the methods `then` and `catch`, respectively. Two promise functions are particularly relevant:

- `Promise.all([`p_1`,` p_2`, ...,` p_n`])` returns a promise that either fulfills when all promises $p_1, p_2, \ldots, p_n$ are fulfilled or rejects as the first of them rejects.
- `Promise.race([`p_1`,` p_2`, ...,` p_n`])` returns a promise that either fulfills or rejects as soon as the one of the promises is fulfilled or rejected, respectively.

IV. Approach

The NodeRacer approach is composed of three main phases. In Phase 1, it instruments the application and collects information relevant for the following phase from a run, driven by a test or by a manual interaction with the application. Phase 2 uses the collected information to infer a happens-before relation between the callbacks. In Phase 3, it re-runs the application a number of times in a special mode that selectively postpones callbacks while respecting the happens-before relation, to expose event race errors.

A. Observation Phase

In the observation phase, NodeRacer instruments the application code to track asynchronous behavior, function calls, and certain native functions. It then runs the application and collects information about the execution in a log file.

First, NodeRacer tracks the lifetime of asynchronous behavior using the Async Hooks[7] functions `init`, `before`, and `promiseResolve`. Function `init` is called every time an asynchronous callback is registered, in which case NodeRacer adds a log entry with a unique callback ID, the ID of the running callback, and the callback type (Immediate, Timeout, TickObject, Promise, or Other). For timeouts, it also registers the delay specified at the registration. Function `before` is called immediately before a callback is executed, in

[7]https://nodejs.org/api/async_hooks.html

which case a log entry is added with the callback ID. These entries are mainly used to distinguish different occurrences of a callback; e.g., `setInterval` defines a callback that may be called several times. Function `promiseResolve` is called when a promise is settled, in which case a log entry is added with the callback ID and the ID of the running callback, allowing for identification of causality among promises.

The Async Hooks API does not directly give us information about which functions are used as callbacks. For this reason we also use the njsTrace[8] library that provides two hook functions, `onEntry` and `onExit`, that allow NodeRacer to add an entry each time a function is called or returned from, respectively, with information about the function (its name if available, file path, and line number) and the ID of the running callback. Finally, we monkey patch the functions `Promise.all` and `Promise.race` to similarly track when they are entered or returned from (with Async Hooks `init` entries in between).

All the logged information is used in the next phase to identify the callbacks and ordering constraints among them.

B. Happens-Before Identification Phase

In the second phase, NodeRacer identifies a happens-before relation [20] for the callbacks observed in the previous phase. Each callback corresponds to one tick of the event loop. Intuitively, one callback e_i happens-before[9] another callback e_j, denoted $e_i \prec e_j$, if e_j causally depends on e_i. To model `Promise.race` we also use a variant of happens-before, written $\{e_{i_1}, \ldots, e_{i_k}\} \prec\!\!\prec e_j$, which means that e_j can execute after at least one of $e_{i_1}, \ldots, e_{i_k}$ has executed.

To identify the happens-before relation, NodeRacer scans the entries of the log file and applies nine rules. Rules #1 to #4 are drawn from the literature and revisit existing knowledge about asynchronous operations in Node.js [14], [21]–[23]. We extend those with Rules #5 to #9 to bring a more accurate modeling of some Node.js operations.

Rule #1: A callback needs to be registered before it can be executed. If e_j is registered during the execution of e_i then $e_i \prec e_j$. The rule is illustrated in the example in Section II: callback `read` is registered during the execution of callback `stat`. For the sample run, we have that $eStat_{ex} \prec eRead_{ex}$ and $eStat_{ep} \prec eRead_{ep}$.

Rule #2: This rule models the case in which a promise is settled. A callback e_i that resolves a promise p needs to run before the callback e_j that is associated with the promise p itself, i.e. $e_i \prec e_j$.

Rule #3: As mentioned in Section III, callbacks may have different types depending on how they were registered; we use the function $type\colon e \rightarrow \{I, N, T, P, O\}$ to denote the type of callback e, where I, N, T, P, and O abbreviate Immediate, nextTick, Timeout, Promises, and Other, respectively.

[8]https://github.com/ValYouW/njsTrace

[9]We use a notion of happens-before that is in the style of e.g. WebRacer [2] and NodeAV [14], which is based on a weaker notion of causality than Lamport's. Intuitively, we consider structural causality only, without involving reads and writes to shared memory, because we want our happens-before relation to be complete with respect to race conditions.

```
23 function foo() {
24     let p1 = new Promise((resolve, reject) => {
25         function asyncp1() { resolve(); } //run async
26     });
27     let p2 = new Promise((resolve, reject) => {
28         function asyncp2() { resolve(); } //run async
29     });
30     let p3 = new Promise((resolve, reject) => {
31         function asyncp3() { resolve(); } //run async
32     });
33     Promise.all([p1, p2, p3]).then(function pall(values) { });
34     Promise.race([p1, p2]).then(function prace(value) { });
35 }
```

Fig. 2. Example with `Promise.all` and `Promise.race`.

Callbacks of types I, N and P are always scheduled following a FIFO discipline, so if e_i is registered before e_j in the same tick and $type(e_i) = type(e_j) \in \{I, N, P\}$, then $e_i \prec e_j$.

Rule #4: For callbacks of types I, N and P, we can establish relations between callbacks registered in different ticks. Assume $type(e_i) = type(e_j) \in \{I, N, P\}$ and e_i and e_j were registered or resolved in different callbacks, $parent_{e_i}$ and $parent_{e_j}$, respectively. If $parent_{e_i} \prec parent_{e_j}$ then $e_i \prec e_j$.

Rule #5: Timeout callbacks also operate following a FIFO discipline, but their delays need to be taken into account. We use $e.t$ to refer to the specific timeout of callback e. If e_i is registered before e_j, $type(e_i) = type(e_j) = T$ and $e_i.t \leq e_j.t$, then $e_i \prec e_j$. If e_i and e_j are registered in the same tick, then under normal circumstances, the one with the smaller timeout will run first, however, there may be a long delay until it is scheduled to run, in which case the other callback will also be enabled [22]. Therefore, if e_i is registered before e_j (possibly in the same tick) but $e_i.t > e_j.t$, no ordering constraints exist between the two callbacks.

Rule #6: Function `setInterval` provokes several instances of callbacks $e_1, e_2, \ldots, e_n$. As the n callbacks are run one after another, we have $e_i \prec e_{i+1}$ for each $1 \leq i \leq n-1$.

Rule #7: Assume promises $p_1, p_2, \ldots, p_n$ are resolved by callbacks $e_1, e_2, \ldots, e_n$ and $p_1, p_2, \ldots, p_n$ are passed as argument to `Promise.all`, and let e_{all} be its resulting callback. In this case, e_{all} will be called only after $e_1, e_2, \ldots, e_n$, so we have $e_i \prec e_{all}$ for each $1 \leq i \leq n$.

Rule #8: Assume promises $p_1, p_2, \ldots, p_n$ are resolved by callbacks $e_1, e_2, \ldots, e_n$ and $p_1, p_2, \ldots, p_n$ are passed as argument to `Promise.race`, and let e_{race} be its resulting callback. We cannot require $e_i \prec e_{race}$ for each $1 \leq i \leq n$, since only one of them is required to be true in a run, so instead we represent this as $\{e_1, \ldots, e_n\} \prec\!\!\prec e_{race}$.

Rule #9: The high priority of TickObject callbacks allows us to relate them to other types of callbacks. Assume e_i is a TickObject callback registered in $parent_{e_i}$ and e_j is a callback such that $type(e_j) \neq N$. If $parent_{e_i} \prec e_j$ then $e_i \prec e_j$.

In addition to these nine rules, $\prec$ is transitive: if $e_1 \prec e_2$ and $e_2 \prec e_3$ then $e_1 \prec e_3$.

The example in Figure 2 involves six callbacks: *foo* (line 23), *asyncp1* (line 25), *asyncp2* (line 28), *asyncp3* (line 31), *pall* (line 33), and *prace* (line 34). Callback *pall* will execute only after callbacks *asyncp1*, *asyncp2*, and *asyncp3* resolve promises `p1`, `p2`, and `p3`, respectively. Then, we can

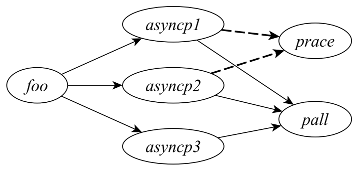

Fig. 3. hb-graph for the example in Figure 2.

Algorithm 1: Instrumentation of a callback function

```
36 Function <FUNCTIONNAME>(<PARAMETERS>)
37     f ← noderacer.getCurrentFunctionInfo()
38     {postpone, $e_i$} ← noderacer.shouldPostpone(f)
39     if postpone then
40         noderacer.postpone( () => {
41             noderacer.notify(f, $e_i$)
42             <RUN ORIGINAL FUNCTION CODE>
43         } )
44     else
45         noderacer.notify(f, $e_i$)
46         <RUN ORIGINAL FUNCTION CODE>
47     end
```

infer by Rule #7 that $asyncp1 \prec pall$, $asyncp2 \prec pall$, and $asyncp3 \prec pall$. Callback $prace$ will depend on which promise resolves first, p1 or p2, so $asyncp1$ or $asyncp2$ may happen before $prace$. Using the representation of Rule #8, we have $\{asyncp1, asyncp2\} \prec\!\!\!\prec prace$.

The resulting happens-before relation can be represented as a graph, which we call an *hb-graph*. Figure 3 shows the hb-graph for the code from Figure 2. The hb-graph is a directed graph with a node for each callback and directed edges representing the happens-before relation. Solid lines and dashed lines represent $\prec$ and $\prec\!\!\!\prec$, respectively. The in-degree of a node e, written $indegree(e)$, is the number of incoming edges to e. We omit edges that are implied by transitivity.

C. Guided Execution Phase

In the third phase, NodeRacer instruments the callback functions and uses the happens-before relation to guide new executions towards alternative callback interleavings by selectively postponing callbacks. By taking the happens-before relation into account when selecting which callbacks to postpone, we steer away from interleavings that are infeasible in real executions and thereby avoid false alarms. Also, the happens-before information can reduce the overhead of postponing callbacks needlessly. The evaluation in Section VI shows the importance of this use of the happens-before information compared to a more naive random exploration.

Algorithm 1 captures the essence of the instrumentation NodeRacer performs for callback functions. The guided execution is implemented by functions `shouldPostpone` (line 38) and `notify` (lines 41 and 45). When an instrumented function is called, NodeRacer collects information about the function obtained from Async Hooks (line 37), and passes it to `shouldPostpone` to decide the next step. If it decides to postpone (line 39), NodeRacer sets up for the callback to run in a future iteration of the event loop (lines 40–43) and remember this for later, such that it is only postponed

Algorithm 2: Function `shouldPostpone`

```
48 Function shouldPostpone(f)
49     postpone ← false
50     if f.isCallback then
51         $e_i$ ← hbgraph.find(f)
52         if $e_i \neq null$ then
53             if hbgraph.mayHappen($e_i$) then
54                 if hbgraph.mayPostpone($e_i$) then
55                     postpone ← randomBoolean()
56                 end
57             else
58                 postpone ← true
59             end
60         end
61     end
62     return {postpone, $e_i$}
```

Algorithm 3: Function `notify`

```
63 Function notify(f, $e_i$)
64     if f.isCallback ∧ $e_i \neq null$ then
65         if hbgraph.mayHappen($e_i$) then
66             hbgraph.remove($e_i$)
67         else
68             noderacer.unexpectedOrder($e_i$)
69         end
70     end
```

once. Otherwise, it runs immediately (lines 45–46). When the callback is actually executed, function `notify` (explained below) is invoked.

The functions `shouldPostpone` and `notify` assume the existence of object `hbgraph` that represents a data structure as in Figure 3, obtained from the previous phase. This object is updated every time a callback is executed, in which case the node representing the callback and its outgoing edges are removed from the graph. In Algorithm 2, `shouldPostpone` initially checks if the current function call is related to a callback (line 50). If it is, f is used to find the associated callback e_i in the hb-graph (line 51). If the current call is neither a callback (line 50) nor is in the hb-graph (e_i is null, line 52), we return with `postpone` set to *false*. If e_i exists, we check if it may be executed now without violating the happens-before relation (line 53); this is determined by checking that $indegree(e_i) = 0$. Otherwise, e_i is postponed (line 58), which gives time for its dependencies to be executed before e_i is run in the future. This mechanism ensures that in situations where $e_j \prec e_i$ and e_j has been postponed and has not been executed yet, then e_i is also postponed to make sure we respect the happens-before relation. Line 54 verifies if e_i can be interleaved with other callbacks by checking if there exist other callbacks with no dependencies, i.e. $|\{e \mid indegree(e) = 0\}| > 1$. If e_i is the only one with no dependencies, postponing it would delay all remaining callbacks without increasing the chance of changing their order, which would be a waste of time. If e_i can be interleaved, NodeRacer randomly decides whether to postpone it or not (line 55); the default delay for a postponed callback is 500 ms.

Function `notify` (Algorithm 3) is called right before the callback is run; as discussed, it has no effect on function calls that are unrelated to callbacks in the hb-graph (in which case

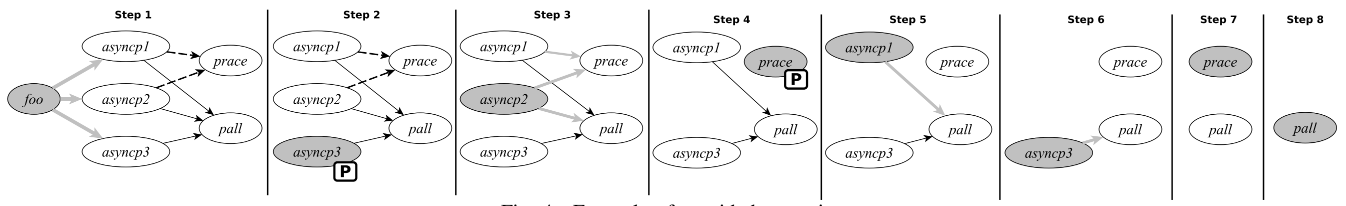

Fig. 4. Example of a guided execution.

isCallback is false in line 64). If callback e_i may happen (line 65), we remove it from the hb-graph along with all its outgoing edges (line 66). If one of the removed edges is part of $\prec\!\!\!\prec$, we also remove other edges in the same set. If e_i may not happen (i.e., despite the delay there are still some dependencies according to the hb-graph), NODERACER registers e_i and its dependencies in the hb-graph as an unexpected callback order (line 68). In our experiments we only observed this in one occasion that was due to a test has nondeterministic inputs (further details in Section VI).

Figure 4 illustrates the guided execution for the example from Figure 2. The callback currently scheduled by Node.js to occur is represented as a gray node. In step 1, *foo* is called and run since it is the only one that may happen; its node and outgoing edges are then removed. In step 2, an asynchronous task ends and its associated callback $asyncp3$ is scheduled. As $asyncp3$ may interleave with other callbacks ($asyncp1$ and $asyncp2$), NODERACER randomly decides to postpone it or not; here $asyncp3$ is postponed (depicted as a box marked **P**). In step 3, $asyncp2$ is scheduled but this time NODERACER decides not to postpone; it is then run and removed from the graph. The edges $(asyncp2, prace)$ and $(asyncp1, prace)$ are removed along with the node. In step 4, $prace$ is scheduled to run but NODERACER decides to postpone it. Then, $asyncp1$ is run in step 5 and $asyncp3$ in step 6; notice that $asyncp1$ could be postponed, while $asyncp3$ had already been postponed once. Finally, $prace$ is run in step 7, and $pall$ in step 8.

Other runs will have different decisions in steps 2–5 and, as a consequence, different callback interleavings will be explored. Due to the delays inserted by NODERACER, we thus obtain a variety of callback interleavings that are unlikely to happen in ordinary testing but may appear in production.

V. IMPLEMENTATION

NODERACER is implemented as a Node.js application with a core module of around 1 600 LoC. It has a command-line interface in which the three phases can be run separately. As mentioned in Section IV-A, the observation phase uses the Async Hooks API and the njsTrace library. A modified version of njsTrace is also used for the guided execution phase.

NODERACER includes some functionality to support diagnosis and debugging of race issues. Report facilities include trace files generated for each run in the observation and guided execution phases, as well as images for hb-graphs. Besides, NODERACER can be run in a diagnosis mode where it systematically postpones only one callback per run, using the history of previous runs to make sure a different callback is postponed in each run. Whenever a bug can be reproduced in this mode, we highlight that single callback in the hb-graph as definitely related to the bug.

VI. EVALUATION

We set out the following research questions:

RQ1 How does NODERACER compare with the state-of-the-art fuzzer Node.fz?

RQ2 Does the use of happens-before relations prevent infeasible executions and avoid needless delays?

RQ3 Can NODERACER help diagnose open issues related to races in Node.js applications?

RQ4 Can NODERACER detect previously unknown race bugs or flaky tests using existing test suites?

A. Experimental Setup

To answer **RQ1**, we selected 11 race bugs reported in previous studies: seven bugs from Davis et al. [15] and four bugs from Wang et al. [9]. We reused the experimental package available from Davis et al., removing the bugs easily revealed by ordinary runs, and the ones we could not reproduce. As for the bugs from Wang et al., we cloned each project's GitHub repository and reversed it to a faulty revision using the information available in the issue. Then, we ran the automated tests. In some cases where the existing tests did not create the callbacks involved in the race, we studied the issue report and designed a test that uses the relevant callbacks but does not reveal the race error in ordinary runs without NODERACER. For each of the 11 benchmarks, we thereby have a test that passes in ordinary runs but fails when the race is exercised.

Using this sample of race bugs (see Table I), we compared NODERACER with state-of-the-art fuzzer Node.fz with its default parameterization [15]. We measured how likely each tool is capable of finding the bug in 100 runs, also called the bug reproduction ratio [15]. We also collected how many runs until the first time the test fails. As both tools involve some randomness, we repeated this process 30 times.

In **RQ2**, we implemented on top of NODERACER a naive approach, named NR-naive, that does not use happens-before relations to decide when postponing callbacks. For each callback, NR-naive has a 50/50 chance of postponing it to be executed after a random delay of 0 to 500 ms. By comparing NR-naive and NODERACER we can measure the benefits of NODERACER's happens-before guided approach. For each benchmark in Table I, we ran NR-naive 100 times. To know if NR-naive provokes infeasible interleavings, we counted how many runs have at least one happens-before violation. We also collected the number of test case failures and how many of them have some happens-before violation. Finally,

TABLE I
BENCHMARKS.

ID	Project Name	Issue[1]	LOC[2]	Description
#1	*agentkeepalive*	23	1.8K	Enhancement features for keepAlive and HTTP agent.
#2	*fiware-pep-steelskin*	269	6.1K	Policy enforcement point proxy for FiWare components.
#3	*Ghost*	1834	30K	Platform for building and running publications (e.g., blogs, magazines).
#4	*node-mkdirp*	2	0.2K	Library to recursively create directories.
#5	*nes*	18	3.4K	Native WebSockets plugin for hapi-based application servers.
#6	*node-logger-file*	1	0.9K	File endpoint for logging.
#7	*socket.io*	1862	2.4K	Framework for real-time event-driven communication.
#8	*del*	43	0.2K	API to delete files and directories using globs.
#9	*linter-stylint*	63	0.2K	Linter plugin for the Atom editor.
#10	*node-simplecrawler*	298	3.9K	Flexible event driven API for crawling web sites.
#11	*xlsx-extract*	7	1K	Data extractor for XLSX files.

[1]The IDs have links to the corresponding GitHub issues. [2]Number of lines of JavaScript code according to cloc.

we measured the average number of needlessly postponed callbacks. (A postponed callback e is needless when there is no other callback that can currently interleave with e.)

As for **RQ3**, we first obtained two curated lists of open source Node.js applications [24], [25]. For the projects in these lists, we used the GitHub API[10] to search for open issues (not opened by our initiative) that have the keyword 'race' in their title or body. We filtered out the issues that were neither related to race conditions nor involved races within the Node.js platform. As the investigation of an arbitrary open issue involves a lot of effort on program comprehension, tests, and debugging, we limited our analyses to five issues.

Concerning **RQ4**, the curated lists of RQ3 were initially used. To simplify the experiments, we sampled projects that use the test framework Mocha.[11] A few projects that use features currently not supported by NODERACER were also removed (see the main limitations in Section VI-C). For each project, we ran the test suite and removed the failing tests and the ones that cannot run independently (e.g., due to dependency of other tests). We also filtered out the tests that have no callback interleaving in the application code. We used the hb-graph to count how many callbacks may interleave with others; if it is zero, the test was removed.

In total, we selected 159 test cases from eight different projects. For each test, we ran NODERACER 100 times. If the test failed in at least one run, we investigated and flagged it as bug, flaky test, or false alarm. For the investigation in RQ3 and RQ4, we used the resources provided by NODERACER: traces, happens-before information, and the diagnosis mode.

A description of procedures, artifacts, and the NODERACER tool are available at https://brics.dk/noderacer.

B. Analysis of Results

RQ1: Figure 5 compares the bug reproduction ratio for each benchmark. We could not run Node.fz with four benchmarks (marked with **).[12] NODERACER has higher reproduction

[10]https://developer.github.com/v3

[11]https://mochajs.org

[12]#5 provokes a segmentation fault, #8 fails with an internal error message, and #9 and #11 have syntax errors due to new features of JavaScript. Node.fz is based on an outdated version of Node.js, which also prevents us from running Node.fz with projects used in RQ3 and RQ4.

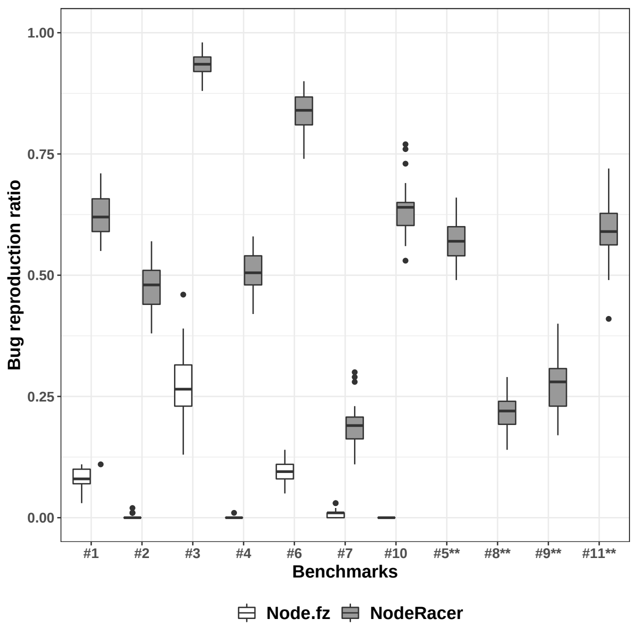

Fig. 5. Bug reproduction ratio. **Node.fz did not run with this benchmark.

ratio in the seven remaining benchmarks. The difference is particularly significant in four cases: in benchmarks #2, #4, and #7, Node.fz finds the bug with very low probability; for benchmark #10, Node.fz cannot reveal the bug at all. In the latter case, the bug depends on a request that takes some time to process, akin to the motivating example in Section II.

As expected, the same trend shows in the number of runs until the first time each test fails. Again, NODERACER has the best results, revealing the bug in less than 5 runs on median.

With this, our answer to RQ1 can be summarized as follows.

NODERACER has a better bug reproduction rate and reveals bugs with fewer runs than Node.fz. Moreover, NODERACER runs on more benchmarks and can uncover a bug not revealed by Node.fz.

RQ2: Table II summarizes the results for NR-naive. The second column shows the percentage of runs that have at least one happens-before violation. Notice that a run with

TABLE II
RESULTS FOR NR-NAIVE (RQ2).

Benchmark	RwHBV[1]	Failures[2]	Needlessly postponed[3]
#1	99%	54 / 55 (98.2%)	2.6 / 18 (14.4%)
#2	93%	41 / 41 (100%)	3.6 / 44 (8.2%)
#3	62%	60 / 93 (64.5%)	1.9 / 16 (11.8%)
#4	45%	31 / 50 (62.0%)	0.7 / 8 (9.0%)
#5	15%	10 / 50 (20%)	1.4 / 12 (11.7%)
#6	100%	71 / 71 (100.0%)	4.2 / 25 (16.8%)
#7	53%	10 / 35 (28.6%)	1.1 / 22 (5.0%)
#8	58%	25 / 46 (54.3%)	2.2 / 18 (12.3%)
#9	0%	0 / 37 (0.0%)	0.01 / 7 (0.1%)
#10	80%	61 / 77 (79.2%)	1.8 / 17 (10.5%)
#11	100%	44 / 44 (100%)	4.6 / 30 (15.5%)
Min	0%	0%	0.1%
Max	100%	100%	16.8%
Average	64.1%	64.3%	10.5%

[1]RwHBV stands for percentage of runs with happens-before violations.
[2]Failures with H-B violations / number of failures (percentage).
[3]Needlessly postponed callbacks / number of callbacks (percentage).

at least one happens-before violation represents an infeasible run. While for benchmark #9 all runs were feasible, most benchmarks had a significant proportion of runs with happens-before violations: 64.1% on average. The third column shows that infeasible runs are similarly present when a test fails; for instance, benchmark #4 had 31 runs out of 50 failures (62%), with some happens-before violation. The average across all benchmarks is 64.3%. As infeasible runs may lead to false alarms, this high percentage suggests that false alarms are more likely to appear with NR-naive than with NODERACER.

The last column brings the results for needlessly postponed callbacks; for instance, benchmark #11 had an average of 4.6 needlessly postponed callbacks out of 30 callbacks involved (15.5%). Some computational resource may be saved by avoiding them (10.5% on average), yet such numbers are not significant for these benchmarks.

In summary, the results for NR-naive give evidence that:

> The happens-before guided approach of NODERACER prevents infeasible callback interleavings. It also reduces the likelihood of false alarms; more than half of all failures by NR-naive involve a happens-before violation.

RQ3: We now describe the investigation of the open issues, and how NODERACER was employed.

a) Issue 23 of get-port: Get-port is a library that returns an available TCP port; it is used by >130K projects in GitHub and has >1.5M weekly downloads in npm. The issue reports that several tests are run in parallel and a race condition occurs.[13] The race actually happens between the time *get-port* returns an available port and a new server is started. Two or more servers may then try to start using the same port but only one will succeed. Based on code snippets in the issue, we came up with a simple test case that would exercise the race. The test did not hit the error in ordinary executions, but it happened in 17 out of 100 runs with NODERACER. So, we could confirm with a witness run the presence of the race error. In the issue discussion, someone suggested using an asynchronous mutex[14] to avoid the race. We modified our test to include the mutex and applied NODERACER again. This time, the expected error did not show up after 100 runs. In this case, NODERACER provided extra evidence that the proposed fix is good. Finally, the traces and happens-before relations provided us means to suggest a future enhancement of the library. We observed that a retention timeout of the returned port could be developed without locks and races.

[13]https://github.com/sindresorhus/get-port/issues/23

b) Issue 262 of live-server: Live-server is a development server with live reload capabilities; it is used by >34K projects and has >100K weekly downloads. The issue reports a potential race between changes in a given file served, causing clients to view outdated versions of the file.[15] As no test was provided, we set up a test that starts *live-server*, modifies an HTML file twice, and observes the file served from the client-side perspective. Using NODERACER, this test did not fail even after 100 runs. After providing our feedback that rejected the aforementioned case, a user pointed out to a scenario where an entire directory was deleted and recreated. So, we added two new tests in which a file and a directory are deleted and recreated, respectively. With these tests, we could confirm the bug using NODERACER. Moreover, NODERACER revealed that a potential fix mentioned by the user was insufficient, as also suspected by that user.

c) Issue 1449 of Bluebird: Bluebird is a promise library; it is used by >3.2M projects and has >13.7M weekly downloads. The issue is related to an extension that allows to set up a timeout for a promise to be fulfilled; if the promise is not settled after a delay, the promise is rejected with a TimeoutError. The combination of this extension with `setTimeout` has brought an inconsistent behavior, i.e., the TimeoutError was not intuitively expected.[16] The test provided in the issue reproduces the error even in many ordinary executions. In this case, we used NODERACER to show that a correct output may also be produced for the test (28/100 times) confirming that it is a race bug. While a solution was initially dismissed due to the inconsistent behavior of `setTimeout`, a fix was proposed.[17] With the fix, the test did not fail anymore using ordinary executions or NODERACER. During our analysis, we noticed that the race arises when a timeout occurs. So, we slightly modified the test so that the timeout is now expected. NODERACER caused this test to fail in 34 out of 100 runs. Therefore, programmers may still be misled when combining *bluebird* promise's timeout and `setTimeout`.

d) Issue 3536 of Express: Express is a framework to implement RESTful APIs; it is used by >4.8M projects and has >10M weekly downloads. The issue is about returning a prettified JSON response for a request that has a given parameter. The first solution involved a change per request in the global variable representing the *express* application.

[14]https://github.com/DirtyHairy/async-mutex
[15]https://github.com/tapio/live-server/issues/262
[16]Open issue #1449, described in https://github.com/petkaantonov/bluebird/issues/1417
[17]https://github.com/petkaantonov/bluebird/pull/1449

TABLE III
RESULTS FOR EXISTING TESTS (RQ4).

Project	#TCs	Fail-TCs	Bugs	Flaky	FAs
bull	6	0	0	0	0
markdown-it	2	0	0	0	0
mongo-express	4	4	4 (2)	0	0
nedb	21	2	0	2	0
node-archiver	23	1	1 (1)	0	0
node-http-proxy	24	0	0	0	0
node-serialport†	60	0	0	0	0
objection.js	19	1	0	0	1
Total	159	8	5 (3)	2	1

†We used Babel to transpile portions of code with async-await.

The code snippet for it was then claimed to be subjected to race conditions. Based on this, we designed a small *express* application with a test that sends two normal requests and one request for prettified JSON, and then checks the format of the three responses. The test was not able to hit the error even after 100 ordinary executions, but it happened in 62 out of 100 runs with NODERACER, thereby confirming that this solution was subjected to races. We noticed that the cause of the race error is not in the library but in the user's code snippet, which modifies a global variable from the callbacks. We then adopted NODERACER to test an alternative solution that formats the output locally per request. After 100 runs and no errors, we had extra evidence that this solution is free of similar races.

e) Issue 3358 of socket.io-client: Socket.io-client is the client library of real-time framework socket.io; it is used by >1.1M projects and has >3.2M weekly downloads. The issue reports a case where a client does not try to reconnect when two sockets are asynchronously open from the same pool.[18] The test initially provided reproduced the bug in ordinary executions. Using NODERACER, the test passed in 48 out of 100 runs, giving evidence that the bug was actually due to an event race condition. We then tested the submitted fix[19]; the tests passed in all 100 runs.

The investigation of the open issues shows that:

> NODERACER is helpful to diagnose race issues. If there is a suspicious test scenario, it supports the reproduction of different interleavings as well as the understanding of ordering (or lack of) between callbacks.

RQ4: Table III summarizes the results of applying NODERACER to existing test suites; it shows the project, number of test cases (#TCs), number of tests that failed (Fail-TCs), and number of tests that failed due to bugs (unique bugs), flaky tests, and false alarms (FAs). In total, we ran NODERACER in 159 tests that may have callback interleavings, making 8 tests to fail: 5 of them were flagged as bugs (3 unique bugs), 2 were flaky tests, and 1 was a false alarm.

NODERACER uncovered bugs in projects *mongo-express* and *node-archiver*. The 4 failing tests in *mongo-express* were related to 2 unique bugs. For the first bug, we submitted an issue.[20] We noticed that this race had been previously detected and partially fixed with a timeout.[21] While the timeout fixed the race for trivial runs, the bug remains and was exercised by NODERACER. The second bug made the test produce an HTTP 500 (Internal Server Error).[22] Basically, a request may arrive before a database-related callback is performed, causing the use of an undefined variable. For *node-archiver*, NODERACER uncovered a previously unknown bug; its essence is illustrated by the motivating example in Section II. A pull request with a bug fix was submitted including a test that consistently reveals the bug in ordinary executions.[23]

There are 2 flaky tests in *nedb*; both are related to the use of function `ensureIndex`. The tests incorrectly assume that the function is synchronous and sometimes fail due to unexpected callback ordering. We fixed the ordering by passing a callback to the function and verified the fix with NODERACER; a pull request was submitted.[24] We also found an inconsistency in the documentation, stating that this function is synchronous. We opened an issue about it as well.[25]

The false alarm in *objection.js* was caused by a nondeterministic test that sets up random timeouts. By modifying the timeouts randomly, the happens-before relations initially observed are violated during the guided execution. We replaced the timeouts by I/O operations (preserving the potential interleavings); for this case, NODERACER ran without failures.

This exploratory study showed that:

> If the existing tests create scenarios with potential interleavings, NODERACER can exercise them to reveal previously unknown bugs and flaky tests.

C. Limitations and Threats to Validity

A first threat to validity of our conclusions is the representativeness of benchmarks. Regarding selection of bug for the study, we opted by issues collected and confirmed by previous studies. They all represent bugs caused by races which were reported, verified, and fixed in their project repositories. All benchmarks are real-world Node.js applications collected from GitHub. Most of them are active projects and have many users and npm downloads.

Implementation flaws may be a potential threat. To mitigate them, we performed two main tasks. First, NODERACER has been extensively tested with a micro-benchmark of 35 small programs that simulate trivial uses (and corner cases) of asynchronous operations in Node.js. Second, we verified sample runs of the 11 benchmarks used of RQ1 and RQ2, as well as checked the results for 13 projects of RQ3 and RQ4.

Node.fz has parameters that can be tuned to improve its effectiveness. We surmise that tuning Node.fz for each case is not practical, so we adopted its default setting. By doing this,

[18]https://github.com/socketio/socket.io/issues/3358
[19]https://github.com/socketio/socket.io-client/pull/1253

[20]https://github.com/mongo-express/mongo-express/issues/499
[21]https://github.com/mongo-express/mongo-express/pull/320
[22]https://github.com/mongo-express/mongo-express/issues/500
[23]https://github.com/archiverjs/node-archiver/pull/388
[24]https://github.com/louischatriot/nedb/pull/610
[25]https://github.com/louischatriot/nedb/issues/609

Node.fz did not reveal all the bugs and performed poorly in some benchmarks, although in principle it can find them (or be more effective) with the right parameters.

The approach can miss happens-before relations introduced by, e.g., third-party libraries or Node.js addons. This may trigger interleavings that are infeasible and potentially cause false alarms, though we did not observe any in the experiments.

Our implementation relies on several third-party libraries, being then subjected to their limitations. For instance, the current version of njsTrace does not support all features of JavaScript, and it fails for some corner cases, breaking the observation and guided execution phases.

NODERACER is designed to work with one process, which can be expected to cover around 95% of all concurrency bugs [9]. Future extensions could be added to detect event races that involve the communication among multiple processes and the experimental API worker threads. NODERACER may also miss races that manifest outside of Node.js; for example, a race bug may arise in an external service when it receives two concurrent requests from a Node.js application.

Finally, the effectiveness of dynamic race detection depends on runs that exercise "interesting" scenarios. Unfortunately, only a small part of existing test suites produce any event scheduling nondeterminism. For this reason, as future work it may be interesting to investigate approaches to synthesize tests that trigger races [26] and exploit client tests [11].

VII. RELATED WORK

Race conditions may provoke serious issues in different kinds of concurrent software systems [27]. This topic has been extensively researched in multi-threaded programs [28]–[32], though races may still occur in event-driven systems [33]. The remainder of this section focuses on races in JavaScript.

Client-side JavaScript. Several approaches have been proposed for race detection in client-side JavaScript. Zheng et al. [1] propose a static analysis to detect races related to inconsistency and atomicity violations in AJAX interactions. The WebRacer [2] and EventRacer [12] tools are built on top of the browser framework WebKit to collect dynamic information and report races. Most of the reported races are harmless [12], [13], so WebRacer and EventRacer implement countermeasures like post-processing filters and coverage criteria. In the same line, Mutlu et al. [3] target harmful races that flow to persistent states.

Other tools go beyond predicting races in the pursuit of a witness run [4]–[7], [13]. WAVE [13] and R^4 [7] adopt an instrumented version of WebKit so that alternative event sequences are explored in a controlled execution. In particular, R^4 employs a technique called conflict-reversal bounding to minimize the event reordering; this has inspired the diagnosis mode of NODERACER. RClassify [4], InitRacer [5], EventRaceCommander [8], and AjaxRacer [6] diverge from previous tools in the implementation design. Instead of adopting some kind of browser modification, they instrument the client-side JavaScript code. Their authors argue that this platform-agnostic design is more robust to changes and updates than platform-specific solutions. The idea of postponing events in NODERACER is also used by EventRaceCommander [8] and AjaxRacer [6], for repairing event race errors and for identify harmful races between AJAX events, respectively.

Node.js. Parts of our happens-before modeling are based on previous work on modeling of asynchronous behaviors in Node.js [21]–[23]. Only a couple of techniques target races in Node.js applications: NodeAV and Node.fz.

NodeAV [14] focuses on detecting races classified as atomicity violations. From a trace, the tool predicts atomicity violations by identifying happens-before relations, a supposedly-atomic pair of events, and violation patterns. The tool adopts Async Hooks to track asynchronous behavior and Jalangi [34] to collect operations on shared resources. NodeAV is tailor-made for a specific class of atomicity violations and cannot detect other kinds of races. Besides, its predictive approach is subjected to false alarms like WebRacer and EventRacer. We could not compare with NodeAV since the tool and its benchmarks are not available.

Node.fz [15] is a fuzzing tool that enables exploration of the callback scheduling. To do so, it takes control of the Node.js' internal event queues, injects very small delays so that the queues are filled with enough events, and shuffles them before each callback is executed. As explained in the introduction, this approach has some drawbacks. By only shuffling callbacks that are ready to run, it misses bugs that only show up when specific events are postponed sufficiently. We discuss this issue in the motivating example, and the experimental results showed that Node.fz performed poorly compared to NODERACER. In theory, Node.fz can perform much better, though it would require nontrivial manual tuning for each case to be effective. Another limitation of Node.fz is that it limits the size of the worker pool to one thread, which prevents it from reaching all possible interleavings when multiple workers are involved. Node.fz is implemented by modifying the internals of the outdated Node.js v0.12.7, making it incompatible with many modern JavaScript projects.

VIII. CONCLUSION

We have presented the NODERACER approach to uncover event race errors in Node.js applications. From an observed run, a happens-before relation is identified and used to perform guided re-runs in which callbacks are selectively postponed to explore different interleavings. NODERACER is implemented entirely in JavaScript, without changes to the Node.js platform.

The experimental evaluation shows that NODERACER outperforms the state-of-the-art fuzzer Node.fz, revealing race bugs with higher probability and with fewer re-runs, while finding bugs that cannot be revealed by the default setting of Node.fz. The happens-before guided execution avoids infeasible interleavings, needlessly postponed callbacks, and false alarms. Our approach can also support the diagnosis of unsolved issues related to race conditions. Finally, NODERACER can help to uncover previously unknown bugs and flaky tests in existing test suites.

References

[1] Y. Zheng, T. Bao, and X. Zhang, "Statically locating web application bugs caused by asynchronous calls," in *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*. ACM, 2011, pp. 805–814.

[2] B. Petrov, M. T. Vechev, M. Sridharan, and J. Dolby, "Race detection for web applications," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. ACM, 2012, pp. 251–262.

[3] E. Mutlu, S. Tasiran, and B. Livshits, "Detecting JavaScript races that matter," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. ACM, 2015, pp. 381–392.

[4] L. Zhang and C. Wang, "RClassify: Classifying race conditions in web applications via deterministic replay," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE / ACM, 2017, pp. 278–288.

[5] C. Q. Adamsen, A. Møller, and F. Tip, "Practical initialization race detection for JavaScript web applications," *PACMPL*, vol. 1, no. OOPSLA, pp. 66:1–66:22, 2017.

[6] C. Q. Adamsen, A. Møller, S. Alimadadi, and F. Tip, "Practical AJAX race detection for JavaScript web applications," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 2018, pp. 38–48.

[7] C. S. Jensen, A. Møller, V. Raychev, D. Dimitrov, and M. T. Vechev, "Stateless model checking of event-driven applications," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. ACM, 2015, pp. 57–73.

[8] C. Q. Adamsen, A. Møller, R. Karim, M. Sridharan, F. Tip, and K. Sen, "Repairing event race errors by controlling nondeterminism," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE / ACM, 2017, pp. 289–299.

[9] J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei, "A comprehensive study on real world concurrency bugs in Node.js," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. IEEE Computer Society, 2017, pp. 520–531.

[10] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. ACM, 2014, pp. 643–653.

[11] G. Mezzetti, A. Møller, and M. T. Torp, "Type regression testing to detect breaking changes in Node.js libraries," in *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, ser. LIPIcs, vol. 109. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, pp. 7:1–7:24.

[12] V. Raychev, M. T. Vechev, and M. Sridharan, "Effective race detection for event-driven programs," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. ACM, 2013, pp. 151–166.

[13] S. Hong, Y. Park, and M. Kim, "Detecting concurrency errors in client-side JavaScript web applications," in *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*. IEEE Computer Society, 2014, pp. 61–70.

[14] X. Chang, W. Dou, Y. Gao, J. Wang, J. Wei, and T. Huang, "Detecting atomicity violations for event-driven Node.js applications," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 2019, pp. 631–642.

[15] J. C. Davis, A. Thekumparampil, and D. Lee, "Node.fz: Fuzzing the server-side event-driven architecture," in *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. ACM, 2017, pp. 145–160.

[16] S. Tilkov and S. Vinoski, "Node.js: Using JavaScript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, 2010.

[17] J. C. Davis, G. Kildow, and D. Lee, "The case of the poisoned event handler: Weaknesses in the Node.js event-driven architecture," in *Proceedings of the 10th European Workshop on Systems Security, EUROSEC 2017, Belgrade, Serbia, April 23, 2017*. ACM, 2017, pp. 8:1–8:6.

[18] Node.js, "Node.js v12.11.0 documentation," 2019. [Online]. Available: https://nodejs.org/api/

[19] M. Madsen, O. Lhoták, and F. Tip, "A model for reasoning about JavaScript promises," *PACMPL*, vol. 1, no. OOPSLA, pp. 86:1–86:24, 2017.

[20] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[21] M. C. Loring, M. Marron, and D. Leijen, "Semantics of asynchronous JavaScript," in *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages, Vancouver, BC, Canada, October 23 - 27, 2017*. ACM, 2017, pp. 51–62.

[22] H. Sun, D. Bonetta, F. Schiavio, and W. Binder, "Reasoning about the Node.js event loop using Async Graphs," in *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*. IEEE, 2019, pp. 61–72.

[23] T. Sotiropoulos and B. Livshits, "Static analysis for asynchronous JavaScript programs," in *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom.*, ser. LIPIcs, vol. 134. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019, pp. 8:1–8:30.

[24] Sqreen, "Curated list of awesome open-source applications made with Node.js," 2019. [Online]. Available: https://github.com/sqreen/awesome-nodejs-projects

[25] Sindresorhus, "Curating the best Node.js modules and resources," 2019. [Online]. Available: https://github.com/sindresorhus/awesome-nodejs

[26] M. Samak, M. K. Ramanathan, and S. Jagannathan, "Synthesizing racy tests," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. ACM, 2015, pp. 175–185.

[27] F. A. Bianchi, A. Margara, and M. Pezzè, "A survey of recent trends in testing concurrent software systems," *IEEE Trans. Software Eng.*, vol. 44, no. 8, pp. 747–783, 2018.

[28] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.

[29] C. Boyapati and M. C. Rinard, "A parameterized type system for race-free Java programs," in *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14-18, 2001*. ACM, 2001, pp. 56–69.

[30] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," in *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*. ACM, 2006, pp. 308–319.

[31] K. Sen, "Race directed random testing of concurrent programs," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. ACM, 2008, pp. 11–21.

[32] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. ACM, 2009, pp. 121–133.

[33] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C. Staicu, "A survey of dynamic analysis and test generation for JavaScript," *ACM Comput. Surv.*, vol. 50, no. 5, pp. 66:1–66:36, 2017.

[34] K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for JavaScript," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. ACM, 2013, pp. 488–498.