NodeRT: Detecting Races in Node.js Applications Practically

Jingyao Zhou
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
im.zjy@smail.nju.edu.cn

Lei Xu*
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
xlei@nju.edu.cn

Gongzheng Lu
Suzhou City University
Suzhou, China
lugz@szcu.edu.cn

Weifeng Zhang
Nanjing University of Posts & Telecommunication
Nanjing, China
zhangwf@njupt.edu.cn

Xiangyu Zhang
Purdue University
West Lafayette, USA
xyzhang@cs.purdue.edu

ABSTRACT

Node.js has become one of the most popular development platforms due to its superior concurrency support. However, races induced by the nondeterministic execution order of event handlers may occur in Node.js applications, causing serious runtime failures. The state-of-the-art Node.js race detector NRace builds a happens-before (HB) graph before detection with a set of HB relation rules. In detection, NRace utilizes a heavy-weight BFS-based algorithm to query the reachability between resource operations, which introduces substantial overhead in practice, causing NRace inapplicable to real-world Node.js application test processes. This paper proposes a more practical Node.js dynamic race detection approach called *NodeRT* (**Node**.js **R**ace **T**racker). To reduce unnecessary overhead, NodeRT simplifies the HB relation rules, and divides the detection into three stages: *trace collection stage*, *race candidate detection stage*, and *false positive removal stage*. In the trace collection stage, NodeRT constructs a partial HB graph called *asynchronous call tree* (ACTree), enabling efficient reachability queries between event handlers. In the race candidate detection stage, NodeRT performs detection on the ACTree, which effectively eliminates most non-racing event handlers and outputs race candidates. In the false positive removal stage, NodeRT utilizes *matching rules* derived from HB relation rules and features of resources to reduce false positives in the race candidates. In experiments, NodeRT detects all known races and 9 unknown harmful races in real-world applications, whereas NRace only detects 3 of the unknown harmful races, with 64× more time consumption on average. Compared with NRace, NodeRT significantly reduces the overhead, making it practical to be integrated into real-world test processes.

*Corresponding Author

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Node.js, race detection, event-driven architecture

ACM Reference Format:
Jingyao Zhou, Lei Xu, Gongzheng Lu, Weifeng Zhang, and Xiangyu Zhang. 2023. NodeRT: Detecting Races in Node.js Applications Practically. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598139

1 INTRODUCTION

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine, providing APIs such as *os*, *file system*, and *http* [15], to enable the development of JavaScript CLI, server-side, and desktop applications. Node.js has become one of the most popular development platforms. The official Node.js package manager *npm* is the world's largest software registry [7]. Node.js is also widely used in the industry. Giants such as Amazon, Netflix, and PayPal have adopted Node.js as their back-end platform [23].

Node.js implements the JavaScript event-driven architecture [28] by utilizing *libuv*, which is composed of one execution thread and six handler queues with different priorities [13, 41], to support concurrency. When an asynchronous call, e.g., an I/O function call, is made by the execution thread, an event e containing a handler $e.handler$ is registered, and a task t, e.g., making an I/O system call, is delegated to *libuv*. When t completes, e.g., the I/O system call returns, *libuv* triggers e, pushing $e.handler$ into one of the handler queues according to the type of e ($e.type$) [14]. The execution thread continuously *polls* the handler queues and executes event handlers in them [41]. In Node.js, tasks do not block the execution thread [7] since tasks are *concurrently* processed by other dedicated threads managed by *libuv*, enabling Node.js to support concurrency.

As the completion times of tasks can be determined by external factors (e.g., network latency), the execution order of event handlers can be nondeterministic. Suppose two event handlers with a nondeterministic execution order access the same resources, such as variables or files, the final states of the resources are uncertain and may not meet developers' expectations. Such races are challenging to discover or reproduce by standard test processes, since they are often triggered by strict invocation combinations and have specific timing requirements, which are rarely considered and satisfied in the test environment. Races in Node.js can cause severe failures, including unexpected application outputs, corrupted data in files or databases, and crashes [43]. Hence, Node.js developers can substantially benefit from practical race detectors.

With the prevalence of Node.js, race detection approaches for Node.js applications have been proposed. Node.fz [9] and NodeRacer [10] adopt fuzzing techniques, trying to expose races by shuffling the execution order of event handlers. NodeRacer additionally guides the shuffling by utilizing HB relation to eliminate false positives. The inherent randomness of fuzzing techniques determines that both of them can only explore a limited subspace of possible execution orders.The entailed re-executions also consume a significant amount of time [7]. Additionally, Node.fz and NodeRacer identify races by looking for observable effects [9, 10], so they produce false negatives, e.g., issue #13 in *write*, which corrupts files but do not trigger any failure. NodeAV [6] and NRace [7] detect races by using HB graphs. NodeAV focuses on detecting atomicity violations and cannot detect other types of races. NRace, the state-of-the-art Node.js race detector, proposes HB relation rules for Node.js, constructs a full-fledged HB graph, and predicts races by checking the reachability of resource operations on the HB graph. For a set of resource operations $Op(R) = \{op_i | op_i.target = R\}$, NRace iterates through all pairs of resource operations (op_i, op_j) in $Op(R)$ and verifies the reachability between them. However, it is notable that in real-world applications, the majority of pairs in $Op(R)$ should be reachable; otherwise, race failures would occur much more frequently, which does not conform to the reality. As a result, the detection method employed by NRace involves numerous unnecessary reachability checks using a resource-intensive BFS-based algorithm, leading to significant costs. The detection algorithm utilized by NRace has a time complexity of $O(n^2(n+h))$, where n and h denote the number of resource operations and HB relations between the resource operations, respectively. The sheer time complexity makes NRace impractical for use in testing real-world Node.js applications due to the excessive overhead.

In this paper, we propose a more practical dynamic race detection approach for Node.js applications and implement it as a prototype *NodeRT*. NodeRT simplifies the HB relations to reduce overhead. Furthermore, the detection is divided into three stages to minimize costs. Specifically, first, NodeRT supplements the HB relation rules proposed by previous works [7, 10]. These rules are then simplified as relations between event handlers and grouped into three categories: *asynchronous call ($HB_{asyCall}$) rules*, *event priority (HB_{ePri}) rules*, and *event registration order (HB_{eReg}) rules*. Second, NodeRT divides the detection into three stages: *trace collection stage*, *race candidate detection stage*, and *false positive removal stage*. During the trace collection stage, a partial HB graph called *asynchronous call tree* (details in Section 3.3), which enables efficient reachability queries between event handlers, is constructed using $HB_{asyCall}$rules. Subsequently, in the race candidate detection stage, NodeRT performs detection on the ACTree, which effectively eliminates most non-racing event handlers, and outputs race candidates. Since false positives may be present among the race candidates, in the false positive removal stage, NodeRT utilizes a set of *matching rules*, which are derived from HB_{ePri}, HB_{eReg}rules and features of some resources, to reduce false positives. Compared with NRace, the time complexity of NodeRT is reduced to $O(ne(e+h'))$, with n, e and h' being the number of resource operations, event handlers, and HB relations between the event handlers, respectively. Note that $e << n$ and $h' << h$.

We evaluate NodeRT in three aspects. We collect 9 known races in real-world Node.js applications and NodeRT detects all of them. In addition, it discovers 2 unknown harmful races (which have been confirmed) in the applications and helps identify a mistake by developers on the root cause of a known race. To further evaluate its efficacy and practicality, we apply NodeRT to 13 popular real-world Node.js applications covering most Node.js usage scenarios and compare it with NRace. NodeRT detects 7 unknown harmful races using their provided test suites with acceptable overhead, whereas NRace only detects 3 of them with 64× more execution time on average. For complicated applications, NodeRT can finish detection in a few minutes, whereas NRace fails to finish detection in an hour.

We make the following contributions.

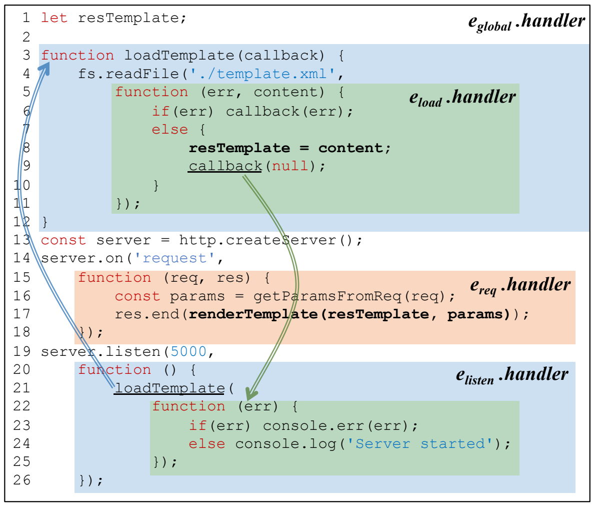
- We propose a dynamic race detection approach for Node.js applications that is more practically applicable to real-world Node.js test processes than the state-of-the-art.
- We implement a prototype NodeRT and evaluate it on real-world Node.js applications. In experiments, NodeRT successfully detects 9 (2+7) unknown harmful races in 5 applications, including popular ones such as *ncp* and *write*. We also identify a mistake on the root cause of a known race using NodeRT. All our findings have been reported to the developers, and 2 of them are confirmed and fixed.
- We compared NodeRT with the state-of-the-art. NodeRT detects 9 unknown harmful races, whereas the state-of-the-art only detects 3 of 9 with at least 64× more execution time on average. NodeRT also shows better compatibility with recent applications.

This paper is structured as follows: Section 2 uses an example to illustrate motivation. Section 3 lists the concepts and symbols used to demonstrate the approach. Section 4 describes the race detection approach. Section 5 describes the implementation of NodeRT, evaluates NodeRT, and compares it with the state-of-the-art. Section 6 discusses the limitations and threats to validity. Section 7 shows the related works, and Section 8 concludes the paper.

2 MOTIVATING EXAMPLE

Figure 1 shows an exemplified real-world race bug [38] detected by NodeRT in *fiware-pep-steelskin*.

In Figure 1, when the application starts, event e_{global} is always triggered first, and $e_{global}.handler$ creates a server (line 13), and invokes `server.on()` to register event e_{req}. The task of waiting for a request is denoted as t_{req} and is delegated to *libuv*. If a request is accepted, t_{req} completes, and e_{req} will be triggered and $e_{req}.handler$ will be put in one of the handler queues for execution. Note that $e_{req}.handler$ generates a response based on variable `resTemplate`. In line 19, the application calls `server.listen()` to bind port 5000 for accepting requests. The call registers event e_{listen}, whose handler is $e_{listen}.handler$ (in lines 20-26 and 3-12). The task t_{listen} that



```
 1 let resTemplate;
 2
 3 function loadTemplate(callback) {
 4     fs.readFile('./template.xml',
 5         function (err, content) {
 6             if(err) callback(err);
 7             else {
 8                 resTemplate = content;
 9                 callback(null);
10             }
11         });
12 }
13 const server = http.createServer();
14 server.on('request',
15     function (req, res) {
16         const params = getParamsFromReq(req);
17         res.end(renderTemplate(resTemplate, params));
18     });
19 server.listen(5000,
20     function () {
21         loadTemplate(
22             function (err) {
23                 if(err) console.err(err);
24                 else console.log('Server started');
25             });
26     });
```

Figure 1: A simplified race in *fiware-pep-steelskin*

tries to bind the port is hence delegated. Upon successful binding, i.e., the server can start accepting requests, t_{listen} completes, so e_{listen} is triggered and $e_{listen}.handler$ is queued. According to lines 20-26, $e_{listen}.handler$ calls `fs.readFile()` at line 4, which delegates a file read task t_{load} and registers event e_{load} (line 4) for the completion of the read. $e_{load}.handler$ (in lines 5-11 and 22-25) initializes variable `resTemplate` (line 8).

As shown in Figure 1, $e_{global}.handler$ registers e_{listen} as well as e_{req}, and $e_{listen}.handler$ registers e_{load}. Therefore, the execution of $e_{global}.handler$ *must* happen before $e_{listen}.handler$ as well as $e_{req}.handler$, and the execution of $e_{listen}.handler$ *must* happen before $e_{load}.handler$. Note that *event handler* is not a synonym of *callback*, as an event handler additionally includes all synchronously called functions inside a callback. For example, in Figure 3, the function in line 5-11 is the callback of `fs.readFile()`, but $e_{load}.handler$ includes the functions in line 5-11 and 22-25.

When a request is accepted, $e_{req}.handler$ generates a response from variable `resTemplate`, which is *expected* to have been initialized by $e_{load}.handler$. However, between the registration (i.e., when t_{load} is delegated) and the trigger of e_{load} (i.e., when t_{load} is completed), a request may be accepted and trigger e_{req} since the server has started accepting requests before, so e_{req} can be triggered *before* e_{load}, which indicates that $e_{req}.handler$ may be executed *before* $e_{load}.handler$, and $e_{req}.handler$ may read `undefined` from `resTemplate` and generate an unexpected invalid response. Therefore, a race occurs between $e_{load}.handler$ and $e_{req}.handler$ on `resTemplate`.

It is challenging to detect and reproduce the race using existing test approaches since the time expected to complete t_{load} is uncertain, and the request must be accepted between the delegation and the completion of t_{load} to trigger the race. In practice, the race may occur when there is substantial concurrency, causing exceptions. Race detection techniques are needed to detect such races predictively during testing. Existing techniques [7, 10] can detect the race if test cases covering the buggy code are given. However, in real-world test processes, all test cases should be executed, and it is impossible for developers to pick the appropriate test cases manually before the bug is detected. The fact makes existing techniques not practically applicable to real-world test processes due to their considerable overhead illustrated in Section 1 and Section 5. Reducing detection overhead is necessary to make the detection more practical to real-world Node.js test processes.

3 CONCEPTS

Some concepts and symbols are used to illustrate the approach of NodeRT (Section 4), which are listed below.

3.1 Resource Operation

We define a *resource operation op* performed by $e.handler$ as a *read* or *write* access to a resource R, with $op \in e.handler$, $op.target = R$ and $op.type = read|write$. As shown in Table 1, NodeRT collects operations on seven types of resources: variable, object, file, array buffer, event emitter, socket, and stream. When a new resource R is created, e.g., a variable definition or object construction, NodeRT logs its type as $R.type$. If $R.type = Variable$ and op initializes R, we denote $R.initializer = op$. If op calls the constructor of R and constructs R, we denote $R.constructor = op$.

For variables, compared with NRace, closures are additionally taken into consideration (Section 4.1) for better precision. For a function named f, its definition and invocations are considered as *write* and *read* operations respectively on a variable named f. For objects, each field of an object is considered separately. For sockets and streams, state-changing APIs affect subsequent *read* operations, such as `socket.close()`. NodeRT also considers different types of streams with operations on resources *stream source* and *stream destination*, if present in the application.

In Node.js, all buffer-like structures are built upon `ArrayBuffers`, which are fixed length arrays containing binary data [16, 29], and the feature is not considered by NRace. Therefore, operations on buffers, e.g., `Buffer` and `TypedArray`, are treated as operations on underlying `ArrayBuffers`. We denote the offsets accessed by $e.handler$ as $e.handler.accOffsets$.

Some file APIs, such as `fs.readFile()`, cause two operations: one when invoked (i.e., a task is delegated), and the other when the event handler is called. We log both of them to detect atomicity violations on files. Moreover, we classify the APIs by their access targets: file content or file state (e.g., access permissions), and log the target into corresponding operations as $op.target = R.content$ or $op.target = R.state$.

APIs related to resource types in Table 1 and are mapped into one or multiple *write* and *read* operations. Compared to previous works which do not support detection on ArrayBuffer and Stream, NodeRT can detect races on more resource types.

3.2 Happens-Before Relation Rules

Since the execution of an event handler is atomic, the HB relation between two resource operations in an event handler always exists. Hence, if there is a lack of HB relation between two resource operations, it can be equivalently determined by the lack of HB relation between two event handlers. Therefore, unlike the HB relation used by previous works, which is defined as a partial relation between *resource operations* [7, 10], the HB relation used by NodeRT is defined

Table 1: Resource Types and Related Operations

Resource Type	Read Operations	Write Operations
Variable	Reading value, function invocation	Writing value, function definition
Object, ArrayBuffer	Reading value from a field/index	Writing value to a field/index
File	Delegating a file access task, reading status or content	Deletion, creation, modifying status or content
EventEmitter	Emitting listeners	Adding or removing listeners
Socket, Stream	Writing or receiving data, e.g., `stream.write()`	Calling state-changing APIs, e.g., `socket.close()`

as a partial relation between *event handlers*, which simplifies the definition of HB relation rules and improves performance without losing any efficacy.

The execution of $e_1.handler$ happens before $e_2.handler$ is denoted as $e_1.handler \longrightarrow e_2.handler$. The HB relation rules are used to recognize HB relations. Based on the observation of real-world Node.js applications, the HB relation rules used by NodeRT are grouped into 3 categories: *asynchronous call ($HB_{asyCall}$) rules*, *event priority (HB_{ePri}) rules*, and *event registration order (HB_{eReg}) rules*.

3.2.1 $HB_{asyCall}$ Rules.

For events e_1 and e_2, if $e_1.handler$ registers/triggers e_2, then $e_1.handler \longrightarrow e_2.handler$ and $HB_{asyCall}$ exists between $e_1.handler$ and $e_2.handler$. Formally, $HB_{asyCall}$ can be recognized by applying following rules.

Rule [Event-Registration]: Given events e_1, e_2, if $e_1.handler$ registers e_2, then $e_1.handler \longrightarrow e_2.handler$.

Rule [Promise-Resolve]: Given events e_1, e_2, and promise p, if $e_1.handler$ resolves p, and the resolution of p triggers e_2, then $e_1.handler \longrightarrow e_2.handler$.

$HB_{asyCall}$ is a cause-and-effect relation between the execution of two event handlers. If $e_1.handler$ (directly or transitively) registers/triggers e_2, then the execution of $e_1.handler$ must happen before $e_2.handler$, and we define $e_1.handler$ *asynchronously calls* $e_2.handler$, denoted as $e_1.handler \longrightarrow_{ac} e_2.handler$ and $e_1.handler \longrightarrow^{d}_{ac} e_2.handler$ if the relation is not obtained transitively.

3.2.2 HB_{ePri} Rules.

In Node.js, TickObject or Promise events have the highest priority. All handlers of events with the two types are executed before executing any other handlers, and the handlers of TickObject events are executed before the ones of Promise events [41]. For events e_1 and e_2, $e_1.handler \longrightarrow_{ac} e_2.handler$ and $e_2.type = TickObject$, because of the highest priority, no handlers of events with other types can be executed between $e_1.handler$ and $e_2.handler$. Therefore, for another event e_3, if $e_1.handler \longrightarrow e_3.handler$, then $e_2.handler \longrightarrow e_3.handler$ and vice versa. Similar principle also applies to Promise events. Formally, HB_{ePri} can be recognized by applying following rules.

Rule [TickObj]: Given events e_p, e_1 and e_2, if $e_p.handler \longrightarrow_{ac} e_1.handler$, $e_1.type = TickObject \neq e_2.type$, and $e_p.handler \longrightarrow e_2.handler$, then $e_1.handler \longrightarrow e_2.handler$.

Rule [Promise]: Given events e_p, e_1 and e_2, if $e_1.type = Promise$, $e_2.type \notin \{TickObject, Promise\}$, $e_p.handler \longrightarrow_{ac} e_1.handler$, and $e_p.handler \longrightarrow e_2.handler$, then $e_1.handler \longrightarrow e_2.handler$.

3.2.3 HB_{eReg} Rules.

Some Node.js APIs can control the enqueueing order (i.e., execution order) of certain event handlers. Formally, HB_{eReg} can be recognized by applying following rules.

Rule [Interval]: If events $e_1, e_2, ..., e_n$ are registered in order by $setInterval()$, then $e_i.handler \longrightarrow e_j.handler$ for $1 \leq i < j \leq n$.

Rule [Timer]: Given events e_1 and e_2, $e_1.type = e_2.type = Timeout$, if e_1 is registered before e_2, the delay time of e_1 is no more than e_2, and exists event e fulfilling $e.handler \longrightarrow^{d}_{ac} e_1.handler$, $e.handler \longrightarrow^{d}_{ac} e_2.handler$, then $e_1.handler \longrightarrow e_2.handler$.

Rule [FIFO]: Given events e_1, e_2, $e_1.type = e_2.type \neq Timeout$, if e_1 is registered before e_2, and exists event e fulfilling $e.handler \longrightarrow^{d}_{ac} e_1.handler$, $e.handler \longrightarrow^{d}_{ac} e_2.handler$, then we have $e_1.handler \longrightarrow e_2.handler$.

Rule [Promise-All]: Given $p_a = Promise.all(p_1, p_2, ..., p_n)$, and the resolutions of $p_a, p_1, p_2, ..., p_n$ trigger $e_a, e_1, e_2, ..., e_n$, respectively, then $e_1, e_2, ..., e_n \longrightarrow e_a$. The rule also applies to similar APIs like `Promise.allSettled()`.

Rule [Promise-Race]: Given $p_r = Promise.race(p_1, p_2, ..., p_n)$, p_i is the first resolved promise among $p_1, p_2, ..., p_n$, and the resolution of p_r and p_i trigger e_r and e_i, respectively, then $e_i \longrightarrow e_r$. The rule also applies to similar APIs like `Promise.any()`.

3.2.4 Discussion.

NodeRT supplements the HB relation rules proposed by previous works [7, 10]. For example, given events e_1, e_2 and e_{1_p}, e_{2_p}, $e_{1_p}.handler \longrightarrow^{d}_{ac} e_1.handler$, $e_{2_p}.handler \longrightarrow^{d}_{ac} e_2.handler$, $e_1.type = e_2.type = Timeout$, previous works only compares the registration times and delays of events *at runtime* to decide HB relation between the resource operations in their handlers. However, if there is no HB relation between $e_{1_p}.handler$ and $e_{2_p}.handler$, the comparison results varies and introduce false negatives. Therefore, $e_{1_p} = e_{2_p}$, i.e., both e_1 and e_2 are registered by the same event handler, is required to make the comparison meaningful for recognizing HB relation between $e_1.handler$ and $e_2.handler$ (Rule [Timer]). The same correction is also applied to derive Rule [FIFO]. Moreover, Rule [Promise] is added to correctly recognize HB relations related to handlers of Promise events.

3.3 Asynchronous Call Tree (ACTree)

An asynchronous call tree is a partial HB graph, whose nodes are event handlers and edges are created by applying $HB_{asyCall}$ rules on event handlers. An ACTree describes asynchronous call relations between event handlers in an application.

Direct asynchronous call relations between event handlers constitutes a tree structure. As derived from $HB_{asyCall}$ rules, in Node.js applications, an event handler (caller) can asynchronously call multiple other event handlers (callees) directly, and a callee can only be asynchronously called by one caller directly. Since all Node.js applications are ignited by a special event handler (called $e_{global}.handler$ in this paper), the direct asynchronous call relations

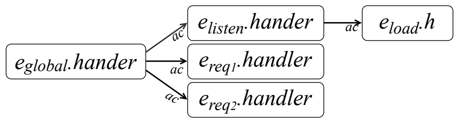


Figure 2: ACTree of the application in Figure 1.

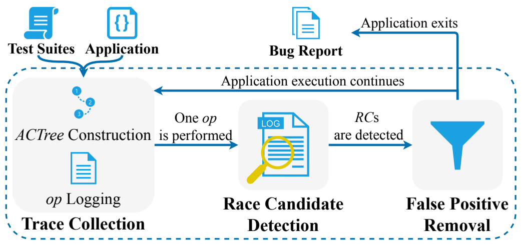


Figure 3: NodeRT Overview. *op* represents *resource operation*.

between event handlers in an application can constitutes an ACTree with $e_{global}.handler$ as its root, and the tree can be used to describe direct and transitive asynchronous call relations between event handlers. For example, Figure 2 shows the ACTree of the application in Figure 1 with two requests accepted. It is evident that an ACTree can achieve superior efficiency in verifying reachability compared to a graph.

Given an ACTree, if $e_1.handler \rightarrow^{d}_{ac} e_2.handler$, then we call $e_1.handler$ as the *parent* of $e_2.handler$.

4 APPROACH

Figure 3 shows the overview of NodeRT. NodeRT detects races on-the-fly by executing the test suites of applications. During execution, in the trace collection stage, NodeRT collects traces (Section 4.1) to construct an ACTree with $HB_{asyCall}$rules and records resource operations. Once a resource operation is recorded, NodeRT suspends the application and proceeds to the race candidate detection stage to detect race candidates (Section 4.2) with the ACTree. Once a race candidate is detected, NodeRT proceeds to the false positive removal stage to check race candidates with a set of matching rules derived from HB_{ePri}, HB_{eReg}rules and features of resource to decide whether it is a false positive (Section 4.3). If not, the race candidate is outputted to the bug report. Then, NodeRT returns to the trace collection stage and resumes the application.

4.1 Trace Collection Stage

The trace collection stage constructs an ACTree, and records the resource operations performed by event handlers, which are implemented by following *execution hooks* intercepting related information when corresponding instructions are executed.

Hook [AsyncInit]: Invoked when an asynchronous call is initialized. The unique identifiers of caller and callee are intercepted.

Hook [AsyncCall]: Invoked when an asynchronous call is performed. The unique identifier of the callee is intercepted.

To construct an ACTree, NodeRT requires two types of information: (1) the asynchronous call relations between event handlers, and (2) the executing event handler when a resource operation occurs. The *async_hooks* module assigns a unique identifier to each

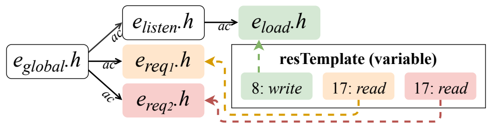


Figure 4: A possible resource trace of variable `resTemplate`. $e.h$ represents $e.handler$.

event handler. Hook [AsyncInit] provides NodeRT with identifier pairs that indicate asynchronous call relations between event handlers. Using this information, NodeRT can create nodes and edges on the ACTree. Hook [AsyncCall] provides NodeRT with the executing event handler, allowing it to create resource operation records and attach them to handler nodes when execution hooks of resource operations are invoked.

Hook [Declare]: Invoked when a variable is declared using `var` or `let`. The executing function where the declaration is in is intercepted for closure analysis.

Hook [ObjLiteral]: Invoked when an object is constructed using object literal.

When a new resource R is created, NodeRT creates a corresponding new resource trace $ResTrace_R$ to contain the records of resource operations performed on R.

Hook [FuncEnterExit]: Invoked when a function starts/ends execution. Used for confirming current executing function.

Hook [FuncLiteral]: Invoked when a function is defined using the `function` keyword or the arrow expression. The executing function where the definition is in is intercepted for closure analysis.

NodeRT performs closure analysis for variables, which is not supported by NRace. Using the information intercepted by Hook [Declare] and Hook [FuncLiteral], NodeRT can perform closure analysis to decide the variable accessed in Hook [Access] more precisely.

Hook [Access]: Invoked when a resource operation *op* is performed on a resource. If the resource is a variable, closure analysis is applied to find its resource trace. A resource operation record $Record_{op}$ is then created and appended to the resource trace.

Hook [SyncCall]: Invoked when a function is synchronously called. Used for track API calls that register event handlers or perform resource operations, e.g., `setTimeout()` and `fs.readFile()`.

For Hook [SyncCall], if a function call registers event e, the register time is recorded as $e.regTime$; if the called function is `setTimeout()` or `setInterval()`, the delay is recorded as $e.delay$.

Hook [AwaitPrePost]: Invoked before or after an `await` expression is executed.

In Node.js, the `await` expression is a mechanism that suspends the executing function until the returned promise is settled [27]. The behavior causes Hook [FuncEnterExit] to be invoked twice on the same function. If not considered, like in NRace, accesses in the same closure will be identified as performed in two different closures, causing imprecise recognition of accessed variables. NodeRT instruments Hook [AwaitPrePost] to resolve the issue.

For example, with the execution hooks, for the application in Figure 1, Figure 4 shows the resource operation records of variable `resTemplate` after two requests are accepted. As shown in Figure 4, given any resource operation, we can identify the handler node $e.handler$ it attaches to and efficiently obtain the portion of ACTree related to $e.handler$, i.e., the path from $e_{global}.handler$ to $e.handler$, which enables efficient reachability queries between event handlers.

4.2 Race Candidate Detection Stage

As defined in previous works [7, 10], a race in Node.js applications is a resource R, and a pair of resource operations op_1, op_2 that are:

(1) **Concurrent:** No HB relation between op_1 and op_2, and
(2) **Conflicting:** Both op_1 and op_2 access R, and at least one of them is write.

As discussed in Section 3.2, the HB relation used by NodeRT is simplified as a relation among event handlers. For NodeRT, the above definition is equivalent to a resource R, and a pair of event handlers $e_1.handler$, $e_2.handler$ that are:

(1) **Concurrent:** $\neg(e_1.handler \rightarrow e_2.handler \vee e_2.handler \rightarrow e_1.handler)$, and
(2) **Conflicting:** $op_1 \in e_1.handler, op_2 \in e_2.handler,$
$(op_1.target = op_2.target = R)$
$\wedge\ (op_1.type = write \vee op_2.type = write)$.

As discussed in Section 1, the method of detecting concurrency in NRace induces substantial cost since it requires reachability checks on mostly non-concurrent resource operations. Therefore, the aim of the race candidate detection stage is to effectively eliminate most non-concurrent event handlers, reducing overhead. To achieve this, NodeRT defines *race candidate* $RC(R, e_1.handler, e_2.handler)$ that is conflicting and *potentially concurrent*:

Potentially Concurrent: $\neg(e_1.handler \rightarrow_{ac} e_2.handler \vee e_2.handler \rightarrow_{ac} e_1.handler)$.

As illustrated in Section 3.3, an ACTree describes asynchronous call relations between event handlers in an application. Therefore, potential concurrency between event handlers can be detected using the ACTree. To detect potential concurrency, as illustrated in Section 4.1, when a resource operation $op \in e_1.handler$ is performed and $op.target = R$, $Record_{op}$ is appended to $ResTrace_R$, and $ResTrace_R$ is then iterated *backwardly* to check if there is any event handler $e_2.handler$ is potentially concurrent with $e_1.handler$. If so, NodeRT checks if any resource operation in $e_1.handler$ is conflicting with $e_2.handler$. Formally, the race candidate detection algorithm is listed in Algorithm 1 and Algorithm 2.

For example, in Figure 4, when the read operation is performed on `resTemplate` by $e_{req_1}.handler$, NodeRT iterates the event handlers in the resource trace (line 8) and finds there is no reachability between $e_{load}.handler$ and $e_{req_1}.handler$ on the ACTree (line 11) using Algorithm 2 (as we can see in Figure 2), indicating potential concurrency. NodeRT then finds that there is a write operation performed on `resTemplate` by $e_{load}.handler$ (line 12), i.e., $e_{load}.handler$ and $e_{req_1}.handler$ are conflicting, so NodeRT detects a race candidate $RC(resTemplate,\ e_{load}.handler,\ e_{req_1}.handler)$ (line 13). Notably, the overhead of Algorithm 2 is much smaller than the BFS algorithm on a graph, as it only involves iterations on at most 2 lists and typically checks a small part of an ACTree. The efficiency of Algorithm 2 can be further optimized by techniques such as chain decomposition [35]. Since the current overhead of Algorithm 2 does not affect the core idea of NodeRT, for simplicity, we leave the optimizations as our future works.

Algorithm 1 Race Candidate Detection

```
Input: $resTrace$ ($ResTrace$)
Output: $RaceCandidate[]$
 1: $groupedOpRecs \leftarrow resTrace.groupOpRecordsByHandler()$
 2: $HANDLER\_NUM \leftarrow groupedOpRecs.length$
 3: $raceCandidates \leftarrow []$
 4: if $HANDLER\_NUM > 1$ then
 5:     $resInfo \leftarrow resTrace.resourceInfo$
 6:     $lastOpGrp \leftarrow groupedOpRecs.getLast()$
 7:     $lastOpGrpEvtHdler \leftarrow lastOpGrp.eventHandler$
 8:     for $i = HANDLER\_NUM - 2$ to 0 do
 9:         $opGrp \leftarrow groupedOpRecs[i]$
10:         $opGrpEvtHdler \leftarrow opGrp.eventHandler$
11:         if $notConn(lastOpGrpEvtHdler, opGrpEvtHdler)$ then
12:             if $conflict(lastOpGrp, opGrp)$ then
13:                 $raceCandidates.push($
14:                 $[resInfo, opGrpEvtHdler, lastOpGrpEvtHdler])$
15:             end if
16:         end if
17:     end for
18: end if
19: return $raceCandidates$
```

Algorithm 2 Reachability Query (*notConn*)

```
Input: $evtHdler_1$ (Event Handler 1), $evtHdler_2$ (Event Handler 2)
Output: Whether the two event handlers are not connected
 1: $parentEvtHdler \leftarrow evtHdler_1$
 2: while $parentEvtHdler \neq null$ do
 3:     if $parentEvtHdler = evtHdler2$ then
 4:         return false
 5:     end if
 6:     $parentEvtHdler = parentEvtHdler.parent$
 7: end while
 8: $parentEvtHdler \leftarrow evtHdler_2$
 9: while $parentEvtHdler \neq null$ do
10:     if $parentEvtHdler = evtHdler1$ then
11:         return false
12:     end if
13:     $parentEvtHdler = parentEvtHdler.parent$
14: end while
15: return true
```

4.3 False Positive Removal Stage

False positives are introduced if only potential concurrency is considered. For $RC(R, e_1.handler, e_2.handler)$, $e_1.handler \rightarrow e_2.handler$ may be introduced by HB_{ePri} and HB_{eReg}, i.e., no concurrency between $e_1.handler$ and $e_2.handler$. In the false positive removal stage, NodeRT utilizes matching rules derived from HB_{ePri}, HB_{eReg} rules and features of some resources, to reduce false positives. The matching rules are listed in Table 2.

In Table 2, the top four rules are *HB matching rules*, which check HB_{ePri} and HB_{eReg} on the event handlers of race candidates, and the bottom four rules are *resource matching rules*, which are derived from features of resources to exclude false positives related to specific resource types. If any match is found, NodeRT deletes

Table 2: Rules for False Positive Matching. $e.h$ is the abbreviation of $e.handler$.

Rule	Name	Rule	Name
$\dfrac{\forall RC(R, e_1.h, e_2.h), \exists e_p, e_1.type = TickObject \wedge e_2.type \neq TickObject \wedge e_p.h \rightarrow_{ac} e_1.h}{RC(R, e_1.h, e_2.h).replacedBy(RC(R, e_p.h, e_2.h))}$	**<TickObj>**	$\dfrac{\forall RC(R, e_1.h, e_2.h), \exists e_3, e_1.type = e_2.type = Timeout \wedge e_3.h \rightarrow^{d}_{ac} e_1.h \wedge e_3.h \rightarrow^{d}_{ac} e_2.h \wedge e_1.delay < e_2.delay \wedge e_1.regTime < e_2.regTime}{BugReport.del(RC(R, e_1.h, e_2.h))}$	**<Timer>**
$\dfrac{\forall RC(R, e_1.h, e_2.h), \exists e_p, e_p.h \rightarrow_{ac} e_1.h \wedge e_1.type = Promise \wedge e_2.type \notin \{TickObject, Promise\}}{RC(R, e_1.h, e_2.h).replacedBy(RC(R, e_p.h, e_2.h))}$	**<Promise>**	$\dfrac{\forall RC(R, e_1.h, e_2.h), \exists e_3, e_1.type = e_2.type \neq Timeout \wedge e_3.h \rightarrow^{d}_{ac} e_1.h \wedge e_3.h \rightarrow_{d} e_2.h \wedge e_1.regTime < e_2.regTime}{BugReport.del(RC(R, e_1.h, e_2.h))}$	**<FIFO>**
$\dfrac{\forall RC(R, e_1.h, e_2.h), R.type = ArrayBuffer \wedge e_1.h.accOffsets \cap e_2.h.accOffsets = \emptyset}{BugReport.del(RC(R, e_1.h, e_2.h))}$	**<Buffer>**	$\dfrac{\forall RC(R, e_1.h, e_2.h), R.type = File, op_1 \in e_1.h, op_2 \in e_2.h, \neg((op_1.target = op_2.target = R.content) \vee (op_1.target = R.state \wedge op_1.type = write) \vee (op_2.target = R.state \wedge op_2.type = write))}{BugReport.del(RC(R, e_1.h, e_2.h))}$	**<File>**
$\dfrac{\forall RC(R, e_1.h, e_2.h), R.type = Variable \wedge op_1 \in e_1.h \wedge op_2 \in e_2.h \wedge (R.initializer = op_1 \vee R.initializer = op_2)}{BugReport.del(RC(R, e_1.h, e_2.h))}$	**<VarInit>**	$\dfrac{\forall RC(R, e_1.h, e_2.h), op_1 \in e_1.h \wedge op_2 \in e_2.h \wedge (R.constructor = op_1 \vee R.constructor = op_2)}{BugReport.del(RC(R, e_1.h, e_2.h))}$	**<Constr>**

($del()$) the race candidate, or replaces ($replacedBy()$) and matches it again. Otherwise, the race candidate is outputted to bug report. Note that $replacedBy()$ does not affect the information in bug report as it remembers the original race candidate, and the replacement is merely for further matching. Before introducing new race candidate $RC(R, e_1.handler, e_2.handler)$, $replacedBy()$ checks if $e_1.handler$ and $e_2.handler$ are not potentially concurrent with Algorithm 2. If they are not potentially concurrent, $replacedBy()$ is equivalent to $del()$, and no new race candidate is introduced.

4.3.1 HB Matching Rules. In Table 2, <Timer> is derived from Rule [Interval] and Rule [Timer] since `setInterval()` uses timers internally. <FIFO> is derived from Rule [FIFO]. As for <TickObj>, in Rule [TickObj], it is imperative to check the HB relation between e_p and e_2 to ascertain the HB relation between e_1 and e_2. To achieve this, <TickObj> replaces e_1 with e_p and conducts another matching process. The same principle also applies to <Promise>.

For example, consider the application in Figure 5, based on $HB_{asyCall}$ rules in Section 3.2.1, we can infer

$$e_{global}.handler \rightarrow_{ac} handler1 \tag{1}$$

$$e_{global}.handler \rightarrow_{ac} handler2 \rightarrow_{ac} handler3 \tag{2}$$

Additionally, since the type of `handler2`'s event is TickObject and `handler1`'s is not, from Rule [TickObj], (1) and (2), we can obtain

$$handler2 \rightarrow handler1 \tag{3}$$

Similarly, since the type of `handler3`'s event is TickObject and `handler1`'s is not, from Rule [TickObj], (2) and (3), we can get

$$handler3 \rightarrow handler1 \tag{4}$$

```
(function schedule() {
  setImmediate(function handler1() {/*..*/});
  process.nextTick(function handler2() {
    process.nextTick(function handler3() {/*..*/});
  });
})();
```

Figure 5: Example of <TickObj>

The matching process of <TickObj> is the inverse derivation of the above process based on Rule [TickObj]. Assuming that the race candidate detection stage outputs $RC(R, handler3, handler1)$, which should be screened by the matching rules. First, since the type of `handler3`'s event is TickObject and `handler1`'s is not, from <TickObj> and (2), the race candidate is replaced by $RC(R, handler2, handler1)$. Then, from <TickObj> and (2), the race candidate should be replaced by $RC(R, e_{global}.handler, handler1)$. However, (1) indicates $e_{global}.handler$ and $handler1$ are not potentially concurrent. Finally, the false positive is excluded from the bug report.

4.3.2 Resource Matching Rules. NodeRT applies four matching rules corresponding to buffers, files, variables and objects, as listed in Table 2. For a buffer R_b from $RC(R_b, e_1.handler, e_2.handler)$, although both of the event handlers access R_b, their accessed parts ($accOffsets$) may not overlap. If so, the race candidate is a false positive and should be excluded, as shown by <Buffer>. For a file R_f, Node.js provides APIs for accessing both its content $R_f.content$ and states $R_f.state$. In $RC(R_f, e_3.handler, e_4.handler)$, only two

Table 3: The Known Races Dataset

ID	Project Name	Issue ID	Commit	Type
1	agentkeepalive	23	`6f059d7`	Socket
2	fiware-pep-steelskin	269	`d1e422d`	EventEmitter
3	nes	18	`3ff3317`	Object
4	node-logger-file	1	`759a2a5`	Object
5	node-mkdirp	2	`b412919`	File
6	simplecrawler	298	`ab1af21`	Object
7	json-file-store	20	`6aada66`	File
8	socket.io-adapter	74	`0667d82`	Object
9	socket.io	4136	`44e20ba`	Map

Table 4: The Exploration Dataset

ID	Project Name	#Stars	#Down	Commit	LoC
1	nodejs-websocket	684	16.6K	`e6a57ed`	689
2	ncp	638	2.2M	`6820b0f`	231
3	baobab	3.1K	2.2K	`a115031`	1687
4	write	81	5.6M	`f537eb6`	90
5	json-fs-store	61	113	`4e75c4f`	90
6	line-reader	440	38.9K	`8b06b2b`	256
7	send	401	348K	`527048b`	125
8	sendfile	43	965	`0fef701`	43
9	serve-static	1.3K	14.6M	`94feefb`	113
10	static	1.1K	323K	`8339e93`	42
11	node-http-proxy	6	863	`9b96cd7`	502
12	WebSocket-Node	3.4K	562K	`a2cd306`	2139
13	json-file-store	189	3.5K	`6aada66`	302

cases that forms races as shown in <File>. First, both of the handlers access $R_f.content$. Second, at least one of the handlers writes to $R_f.state$ as it may affect following accesses to $R_f.content$ and $R_f.state$. <VarInit> and <Constr> exclude the lazy initialization pattern frequently used by Node.js developers.

5 IMPLEMENTATION AND EVALUATION

NodeRT is implemented as a TypeScript CLI tool. NodeRT utilizes the *async_hooks* module and is built upon NodeProf.js [40] for on-the-fly trace collection.

In the evaluation, NodeRT is compared with state-of-the-art Node.js race detector NRace [7], as it shows better efficacy and efficiency than other approaches, e.g., NodeRacer [10]. All experiments were conducted on Intel Core i7-10700F with 16GB memory, running Ubuntu 20.04.3 and Node.js 16.13.0. The revision of NodeProf.js is `8cdc5cc`. NodeProf.js is compiled with GraalVM 21.2.0. All artifacts, including source code, datasets, bug reproducing test cases, bug reports, and original outputs, are all publicly available [39].

To evaluate NodeRT, we set out the following research questions:

RQ1: Can NodeRT detect known races and reason about them?

RQ2: Can NodeRT detect unknown races in real-world Node.js applications, compared with NRace?

RQ3: What is the effectiveness of the matching rules?

RQ4: What is the overhead of NodeRT, compared with NRace?

5.1 Experimental Setup

To answer the research questions, we collected two datasets: the known races dataset, which is for RQ1, and the exploration dataset, which is for RQ2, RQ3 and RQ4.

The known races dataset. We collected 9 known races from NRace [7] and GitHub. NRace provided 10 known races, and we further searched for issues and pull requests on GitHub using keywords such as *race* and *asynchronous bug*. Then, we filtered them using the following criteria: (1) The race can be reproduced. (2) The race has related bug reports that clearly describe its root cause and reproduction method. If the bug reports do not provide any test case, we build one to reproduce the race based on their descriptions. (3) The application is compatible with Node.js 14, which is used by Graal.js [32]. Some known races from NRace, such as *del*, cannot be reproduced. Some others, such as *xlsx-extract*, are too stale to install. Therefore, we excluded them from the dataset. Finally, we obtained 6 known races from NRace (1-6) and 3 from GitHub (7-9), as shown in Table 3. Column *Issue ID* shows the IDs of the issues or pull requests that report the bugs, column *Commit* shows in which commits the bugs is in, and column *Type* shows the types of resources that the races happen on.

The exploration dataset. We collected 13 Node.js applications from GitHub that fulfill the following conditions: (1) The application is compatible with Node.js 14. (2) The application offers asynchronous APIs. (3) The application provides test suites. Once an application is collected, its newest version is downloaded. The detailed information of these applications is listed in Table 4. Column *#Star* shows the counts of stars on GitHub, column *#Down* shows the weekly download numbers of the applications on npm, and column *LoC* shows the lines of JavaScript code in the core modules of the applications. The applications in the dataset cover common Node.js usage scenarios with various sizes, and most of them are prevalent from the perspective of stars and npm downloads.

To answer RQ1, for each application in the known races dataset, we first execute its reproducing test case using NodeRT and obtain bug report 1. We then analyze bug report 1 and source code to confirm that the race is detected and the reported root cause fits the bug reports from GitHub. To validate bug report 1, if the application provide a fix, we download the fixed version and execute the same test case to obtain bug report 2. We compare bug report 1 and 2 to further confirm that NodeRT does detect the known race.

For RQ2 and RQ3, we first modify the test suites of the applications in the exploration dataset, which enables each test case to execute twice, ensuring that every asynchronous API is invoked at least two times. This helps reveal races between two calls of the same API. Second, for each application, we execute the modified test suites with NRace and NodeRT to obtain bug reports. We use Babel to compile the application to ES5 if it is incompatible with NRace. We stop the execution if the detection fails to finish in 1 hour. We then analyze the bug reports, classify races, and summarize runtime statistics. Finally, we compare the detection results and runtime statistics of NRace and NodeRT.

Notably, NRace limits its trace log file to a maximum size of 14075 lines and exceeded traces are discarded by default [8]. To

Table 5: Detection Result on The Known Races Dataset

ID	Project Name	Detected	#HR	#BR	#FP
1	agentkeepalive	Yes	1(0)	4	0
2	fiware-pep-steelskin	Yes	3(2)	29	0
3	nes	Yes	1(0)	0	0
4	node-logger-file	Yes	3(0)	0	0
5	node-mkdirp	Yes	1(0)	0	0
6	simplecrawler	Yes	1(0)	5	0
7	json-file-store	Yes	4(2)	0	0
8	socket.io-adapter	No	0(0)	0	0
9	socket.io	Yes	1(0)	0	0

ensure the detection capability of NRace, we removed the limitation without affecting any core functionality of NRace.

For RQ4, we repeat the execution in RQ2 3 times to obtain average time and memory consumption. Time consumption is measured by the *time* utility. Memory consumption is the maximum memory usage of NodeRT during execution. Memory usage is sampled per second using the `process.memoryUsage()` API of Node.js.

We classify the detected races into 3 categories: *harmful races*, *benign races*, and *false positives*. *Harmful races* make the behaviors of applications deviate from developers' expectations. *Benign races* are races deliberately designed by developers. *False positives* are reported races that do not form race conditions or are impossible to happen in real-world execution. In addition, bug reports from NRace occasionally include inexact or unhelpful race information, e.g., lack of key information such as where the race is located, or the reported racing code is nonexistent in the source code. We ignore them in the evaluation results.

5.2 Detecting Known Races (RQ1)

The detection result on the known races dataset is shown in Table 5. In Table 5, column *#Detected* represents whether the race is detected by NodeRT. Column *#HR* shows the total numbers of harmful races reported by NodeRT, and the numbers of newly detected ones are shown in the brackets. *#BR* and *#FP* indicate the numbers of benign races and false positives, respectively.

As shown in Table 5, NodeRT reports 15 harmful races with no false positive. We merged the 15 harmful races by their causes. The merged result shows that 8 of the 9 known harmful races, and 4 previously unknown harmful races, are successfully detected by NodeRT. The benign races are easy to identify manually by analyzing bug reports from NodeRT. Benign races in *fiware-pep-steelskin* are caused by global cache storage and designed event listener registrations; *simplecrawler* utilizes a shared array as a queue to schedule jobs asynchronously. The result proves that NodeRT can detect races on different types of resources. Except the races that are also detected on the exploration dataset, NodeRT detects 2 unknown harmful races on *fiware-pep-steelskin* as exemplified in Figure 1, which have been confirmed and fixed [38].

As for the undetected race in *socket.io-adapter*, although its developers have confirmed that the bug is caused by races, after analyzing the trace collected by NodeRT, we found out that it is a synchronous bug introduced by *EventEmitter*. We have reported the finding to

Table 6: Detection Result on The Exploration Dataset

ID	NodeRT			NRace		
	#UHR	#BR	#FP	#UHR	#BR	#FP
1	0(0)	5	1	*Timeout*		
2	4(4)	20	0	2	22	0
3*	0(0)	0	0	*Test Error*		
4*	1(1)	0	1	0	0	4
5	1(1)	0	0	1	0	0
6	0(0)	8	0	0	21	0
7*,8*	0(0)	0	0	*Test Error*		
9	0(0)	0	0	*Timeout*		
10*	0(0)	0	0	*Test Error*		
11	0(0)	1	0	0	0	0
12*	0(0)	11	1	*Graph Construction Error*		
13	1(1)	0	1	0	0	0
Total	7(7)	40	4	3	43	4

the developers and proposed fixes [37] to all undiscovered similar bugs. The finding indicates that NodeRT can help uncover the causes of bugs and assist in debugging.

5.3 Detecting Unknown Races (RQ2)

We list the detection result on the exploration dataset in Table 6. *ID** means the application is compiled to ES5 using Babel before instrumented by NRace. Column *#UHR* lists the numbers of harmful races after being merged by unique root causes.

As Table 6 shows, NodeRT detects 7 unknown harmful races on 4 applications including popular applications *ncp* and *write*, while NRace misses 4 of them. NodeRT achieves better efficacy with more supported resource types and supplemented HB relation rules. All the harmful races were not triggered during execution, which demonstrates that NodeRT can predictively detect unknown races. The presence of the harmful races could potentially cause unpredictable failures in a wide range of applications since the applications with them are downloaded millions of times per weak as shown in Table 4. We have reproduced all the harmful races and reported them to developers [39].

NodeRT detects 40 benign races, including races on flag variables, counters, shared event emitters, and cache objects. Similar to RQ1, these benign races can be easily identified based on source codes and bug reports from NodeRT. For example, all the benign races in *ncp* are caused by three flag variables `started`, `finished`, and `running`. It is not difficult for developers to figure out that they are deliberately accessed in different event handlers to identify current execution status based on the source code and bug report.

NodeRT reports 4 false positives, but none of them is produced by mismatches in the the false positive removal stage. The false positive on *json-file-store* is caused by external asynchronous calls introduced by the test framework. The false positive on *write* is caused by imprecise modeling of a Node.js file API. Other false positives are caused by external HB relations, such as network protocol requirements (*nodejs-websocket*) and garbage collection (*WebSocket-Node*).

Table 7: Matching Result on the Exploration Dataset

ID	#HB	#RES	#R	ID	#HB	#RES	#R
1	21,215	123,332	6	8	16	0	0
2	0	1,686	24	9	486	0	0
3	972	0	0	7, 10	0	0	0
4	2,039	0	2	11	0	7	1
5	0	0	1	12	382	2,805	12
6	4	0	8	13	1,866	285	2

While NodeRT finishes detection on all applications, NRace breaks the test suites of 4 applications and raises error on another application after instrumentation even with Babel. NodeRT shows better compatibility with recent Node.js applications than NRace. Furthermore, NRace times out on 2 more applications, making it unable to finish detection on approximately half of the applications in the exploration dataset. NRace is not as capable as NodeRT of detecting races in real-world applications.

5.4 Effectiveness of Matching Rules (RQ3)

Table 7 shows the false positives identified on the exploration dataset. Columns *#HB* and *#RES* represent the numbers of false positives identified by the HB matching rules and the resource matching rules, respectively. Column *#R* shows the numbers of races that constitute the final bug reports.

As the statistics in Table 7 demonstrate, lots of false positives are identified by both the HB matching rules and the resource matching rules, proving that the matching rules are effective in identifying false positives in race candidates. The efficacy of the matching rules is deeply affected by the applications' diversities, that is, the usage variety of asynchronous APIs and resource types.

5.5 Runtime Overhead (RQ4)

The runtime information and overhead statistics are listed in Table 8. Columns *#EH* and *#OP* show the numbers of event handlers and resource operations during the executions of applications. Column *Mem* indicates the peak memory consumption of NodeRT. Columns *Load* and *Detection* show the load time of NodeProf.js and detection time of NodeRT. Columns *Inst*, *TC*, *HBC*, and *DP* show the time consumed by NRace on instrumentation, trace collection, HB graph construction, and race detection, respectively.

As demonstrated in Table 8, the overhead of NodeRT is acceptable on the exploration dataset. All detections finish from seconds to several minutes. With the increase in event handlers and resource operations, the overhead of NodeRT grows. During the experiments, the maximum memory consumption of NodeRT is 2.12GB. Considering the time and effort required to find and reason about race bugs manually, the overhead of NodeRT is well acceptable, and it is practically applicable to real-world Node.js test processes.

NodeRT is at least 64× faster than NRace on average, showing a significant efficiency advantage, which can be explained by the time complexity difference illustrated in Section 1 as the *#OP*s (n) are significantly greater than the *#EH*s (e) as shown in Table 8. NRace is faster only on *json-fs-store*, during which the fewest event handlers and resource operations are processed. With the increase in event handlers and resource operations, the time consumed by NRace inflates. NRace even fails to complete detection on 2 of the 8 compatible applications in one hour. The result shows that the practicality of NRace is significantly limited by its overhead, and NodeRT is more practical for real-world race detection.

6 DISCUSSION

In this section, we discuss some potential threats and limitations in our approach, implementation, and evaluation.

6.1 Threats to Validity

6.1.1 External Threats. The representativeness of experimental applications may be limited under the bounded size of the datasets. There are two primary reasons for this.

The first one is time constraints. Currently, there is no large Node.js race detection dataset exists. As such, we had to manually identify and reproduce races (with an additional focus on excluding races in web applications). These tasks were notably time-consuming. Second, for a fair comparison, we had to ensure that NRace was compatible with the benchmarks. As NRace may not support certain features, our options were further constrained.

We try to mitigate this issue by building our datasets cautiously. All harmful races in the known race dataset have been studied in previous works [6, 10, 43]. For representativeness, our selected applications in the exploration dataset are up to date, covering common Node.js usage scenarios with various sizes, and most of them are popular from the aspect of GitHub stars and npm downloads.

6.1.2 Internal Threats. First, the effectiveness of dynamic analysis is affected by code coverage rate. NodeRT cannot detect races in uncovered code. We try to alleviate this threat by executing application-provided test suites. Most of the test suites reach high code coverage rates. Second, misclassifications of races may exist. We try to minimize the occurrence of misclassification by comparing bug reports from NodeRT and NRace and developing reproducing test cases.

6.2 Limitations

Although our experiments prove that NodeRT is capable of detecting races in real-world test processes, there are still some limitations in our approach and implementation.

Usability limitation. NodeRT is inapplicable to some stale applications using deprecated APIs because Graal.js is designed for executing recent Node.js applications [32].

Incomprehensive support of resource types. In NodeRT, only some frequent resource types are supported, which may prevent NodeRT from detecting more races. Support for more resource types is required to improve the detection capacity of NodeRT.

Soundness issues. The HB relation rules and matching rules used by NodeRT may not thoroughly cover all scenarios appearing in real-world Node.js applications. More studies and experiments are needed to enhance the soundness of our prototype.

NodeRT does not support multi-threaded Node.js applications where 5% of race bugs are discovered [43]. More features are needed to detect races involving processes and threads.

Table 8: Runtime Information and Overhead on the Exploration Dataset

ID	#EH	#OP	NodeRT				NRace					Speedup
			Mem	Load(s)	Detection(s)	Total(s)	Inst(s)	TC(s)	HBC(s)	DP(s)	Total(s)	
1	320	187,646	1.94GB	**5.01**	**51.54**	**56.55**	0.47	1.99	294.79	*3600.00*	*>3897.25*	>68.92×
2	620	18,179	1.30GB	**4.89**	**8.57**	**13.46**	0.63	0.73	1163.74	671.12	1836.22	136.42×
3	421	932,886	2.12GB	**5.61**	**235.82**	**241.43**	*Test Error*					-
4	144	5,262	1.26GB	**4.90**	**6.34**	**11.24**	0.32	1.04	34.45	16.68	54.49	4.85×
5	50	551	1.28GB	4.93	3.45	8.38	**0.32**	**0.23**	**0.43**	**0.29**	**1.27**	0.15×
6	896	12,177	1.35GB	**4.89**	**10.59**	**15.47**	0.30	0.82	108.55	250.70	360.36	23.29×
7	292	18,836	1.29GB	**4.70**	**10.03**	**14.73**	*Test Error*					-
8	33	661	1.42GB	**4.83**	**6.67**	**11.50**	*Test Error*					-
9	409	10,575	1.40GB	**4.84**	**11.74**	**16.57**	0.30	0.98	*3600.00*	-	*>3601.28*	>217.34×
10	148	1,926	1.34GB	**4.74**	**6.39**	**11.13**	*Test Error*					-
11	447	43,542	1.45GB	**4.73**	**19.74**	**24.47**	0.84	4.84	384.49	127.30	517.46	21.15×
12	109	17,628	1.42GB	**4.65**	**10.51**	**15.16**	*Graph Construction Error*					-
13	188	12,021	1.32GB	**4.76**	**9.35**	**14.10**	0.53	1.32	22.75	18.03	42.63	3.02×
Average			1.45GB	4.88	30.06	34.94	0.48	1.56	-	-	-	>64.35×

NodeRT does not support identifying races as benign or harmful currently. More functionalities, e.g., guided replay, are needed to soundly confirm harmful races.

HB relation rules for Promise APIs, e.g., Rule [Promise-All], are not implemented currently due to the lack of Node.js internal information. This limitation may reduce the efficacy of NodeRT.

7 RELATED WORKS

Race bugs exist in various concurrent applications and can result in serious issues [4]. Lots of research has been performed on understanding and detecting race bugs. Section 1 discusses some related works on detecting races in Node.js applications. This section focuses on race detection techniques for other kinds of applications.

Many studies [1, 2, 17, 30, 33, 35, 44, 49] have been conducted on detecting races in client-side JavaScript applications. WebRacer [33] formulates HB relations for web features. WAVE [17] generates sequences of operations and executes them on web applications to detect races. ARROW [44] proposes a static technique that effectively patches certain races on web pages. AjaxRacer [1] combines dynamic analysis and controlled execution to identify AJAX races. Race detection tools for client-side JavaScript applications mainly consider unique features provided by browsers, e.g., DOM and AJAX, which are nonexistent in Node.js. Therefore, it is challenging to apply them to Node.js applications.

Race detection approaches for Android applications [5, 18–20, 25, 36, 45] also focus on detecting races caused by event-driven architecture. CFCA [18] and DroidRacer [25] detect races based on HB graphs. SIERRA [19] reifies threads and events as concurrency actions and infers HB relations inside statically. SIEVE [45] exposes races by selective branch instrumentation. The Android race detection approaches cannot deal with multipriority callback queues in Node.js, making them inapplicable to Node.js applications.

Race detection for multi-threaded applications has been intensively studied, and many techniques have been proposed, including lock-sets [11, 31, 34], dynamic instrumentation [3, 12, 22, 26, 47, 48], May-Happen-in-Parallel relations [21, 50], runtime mitigation [42, 46], and unifying threads and events [24]. Most races in Node.js applications occur within a single thread [43], and Node.js does not support any synchronization mechanism, such as lock or mutex. Thus, applying race detection techniques for multithreaded applications on Node.js applications is challenging.

8 CONCLUSION

We have presented a more practical Node.js dynamic race detection approach and implemented it as a prototype NodeRT. NodeRT supplements HB relation rules for Node.js. For practicality, NodeRT further simplifies the HB relation rules, and divides the detection into three stages: *trace collection stage*, *race candidate detection stage*, and *false positive removal stage*. In the trace collection stage, NodeRT constructs an ACTree, enabling efficient reachability queries. Then in the trace collection stage, NodeRT effectively eliminates most non-racing event handlers by detecting race candidates on the ACTree. Finally, in false positive removal stage, NodeRT utilizes *matching rules* derived from HB relation rules and features of resources to reduce false positives in the race candidates.

The evaluation shows that NodeRT is more practically applicable to real-world Node.js test processes for detecting races than the state-of-the-art. NodeRT can detect unknown races and help reveal their root causes using application-provided test suites with acceptable overhead. Compared to the state-of-the-art, NodeRT shows better efficacy and efficiency, detecting more harmful races with significant speedup.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This research was supported, in part by NSFC (Nos 62272214, 61832009), Cooperation Fund of Huawei-Nanjing University Next Generation Programming Innovation Lab (No. YBN2019105178SW32, No. YBN2019105178SW27). Any opinions, findings, and conclusion in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] Christoffer Quist Adamsen, Anders Møller, Saba Alimadadi, and Frank Tip. 2018. Practical AJAX Race Detection for JavaScript Web Applications. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 38–48. https://doi.org/10.1145/3236024.3236038

[2] Christoffer Quist Adamsen, Anders Møller, and Frank Tip. 2017. Practical Initialization Race Detection for JavaScript Web Applications. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 66 (oct 2017), 22 pages. https://doi.org/10.1145/3133890

[3] Adil Ahmad, Sangho Lee, Pedro Fonseca, and Byoungyoung Lee. 2021. *Kard: Lightweight Data Race Detection with per-Thread Memory Protection.* Association for Computing Machinery, New York, NY, USA, 647–660. https://doi.org/10.1145/3445814.3446727

[4] Francesco Adalberto Bianchi, Alessandro Margara, and Mauro Pezze. 2018. A Survey of Recent Trends in Testing Concurrent Software Systems. *IEEE Trans. Softw. Eng.* 44, 8 (aug 2018), 747–783. https://doi.org/10.1109/TSE.2017.2707089

[5] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2015. Scalable Race Detection for Android Applications. *SIGPLAN Not.* 50, 10 (oct 2015), 332–348. https://doi.org/10.1145/2858965.2814303

[6] Xiaoning Chang, Wensheng Dou, Yu Gao, Jie Wang, Jun Wei, and Tao Huang. 2019. Detecting Atomicity Violations for Event-Driven Node.Js Applications. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, 631–642. https://doi.org/10.1109/ICSE.2019.00073

[7] Xiaoning Chang, Wensheng Dou, Jun Wei, Tao Huang, Jinhui Xie, Yuetang Deng, Jianbo Yang, and Jiaheng Yang. 2022. Race Detection for Event-Driven Node.Js Applications. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, Piscataway, NJ, 480–491. https://doi.org/10.1109/ASE51524.2021.9678814

[8] ChangXiaoning. 2021. *tcse-iscas/nrace.* Technology Center of Software Engineering. Retrieved May 4, 2022 from https://github.com/tcse-iscas/nrace/blob/master/lib/typeerrorDetect/traceParser.js#L97

[9] James Davis, Arun Thekumparampil, and Dongyoon Lee. 2017. Node.Fz: Fuzzing the Server-Side Event-Driven Architecture. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 145–160. https://doi.org/10.1145/3064176.3064188

[10] André Takeshi Endo and Anders Møller. 2020. NodeRacer: Event Race Detection for Node.js Applications. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE Press, Piscataway, NJ, 120–130. https://doi.org/10.1109/ICST46399.2020.00022

[11] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 237–252. https://doi.org/10.1145/945445.945468

[12] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective Data-Race Detection for the Kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 151–162.

[13] OpenJS Foundation. 2021. *The Node.js Event Loop, Timers, and process.nextTick().* OpenJS Foundation. Retrieved December 9, 2021 from https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

[14] OpenJS Foundation. 2022. *Async hooks.* OpenJS Foundation. Retrieved January 27, 2022 from https://nodejs.org/dist/latest/docs/api/async_hooks.html#type

[15] OpenJS Foundation. 2022. *Node.js.* OpenJS Foundation. Retrieved January 27, 2022 from https://nodejs.org

[16] OpenJS Foundation. 2023. *Buffer.* OpenJS Foundation. Retrieved January 17, 2023 from https://nodejs.org/api/buffer.html

[17] Shin Hong, Yongbae Park, and Moonzoo Kim. 2014. Detecting Concurrency Errors in Client-Side Java Script Web Applications. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation (ICST '14)*. IEEE Computer Society, USA, 61–70. https://doi.org/10.1109/ICST.2014.17

[18] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. 2014. Race Detection for Event-Driven Mobile Applications. *SIGPLAN Not.* 49, 6 (jun 2014), 326–336. https://doi.org/10.1145/2666356.2594330

[19] Yongjian Hu and Iulian Neamtiu. 2018. Static Detection of Event-Based Races in Android Apps. *SIGPLAN Not.* 53, 2 (mar 2018), 257–270. https://doi.org/10.1145/3296957.3173173

[20] Yongjian Hu, Iulian Neamtiu, and Arash Alavi. 2016. Automatically Verifying and Reproducing Event-Based Races in Android Apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 377–388. https://doi.org/10.1145/2931037.2931069

[21] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 337–348. https://doi.org/10.1145/2594291.2594315

[22] Jeff Huang and Arun K. Rajagopalan. 2016. Precise and Maximal Race Detection from Incomplete Traces. *SIGPLAN Not.* 51, 10 (oct 2016), 462–476. https://doi.org/10.1145/3022671.2984024

[23] Branko Krstic. 2023. *64 Node JS Stats that Prove Its Awesomeness in 2023.* WebTribunal. Retrieved May 21, 2023 from https://hostingtribunal.com/blog/node-js-stats/

[24] Bozhen Liu, Peiming Liu, Yanze Li, Chia-Che Tsai, Dilma Da Silva, and Jeff Huang. 2021. *When Threads Meet Events: Efficient and Precise Static Race Detection with Origins.* Association for Computing Machinery, New York, NY, USA, 725–739. https://doi.org/10.1145/3453483.3454073

[25] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race Detection for Android Applications. *SIGPLAN Not.* 49, 6 (jun 2014), 316–325. https://doi.org/10.1145/2666356.2594311

[26] Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2021. Optimal Prediction of Synchronization-Preserving Races. *Proc. ACM Program. Lang.* 5, POPL, Article 36 (jan 2021), 29 pages. https://doi.org/10.1145/3434317

[27] MDN. 2022. *async function.* Mozilla. Retrieved April 8, 2022 from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

[28] MDN. 2022. *The event loop.* Mozilla. Retrieved March 29, 2022 from https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

[29] MDN. 2023. *JavaScript typed arrays.* Mozilla. Retrieved January 17, 2023 from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays

[30] Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. 2015. Detecting JavaScript Races That Matter. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 381–392. https://doi.org/10.1145/2786805.2786820

[31] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 308–319. https://doi.org/10.1145/1133981.1134018

[32] Oracle. 2022. *GraalVM JavaScript Implementation.* Oracle. Retrieved January 28, 2022 from https://www.graalvm.org/21.3/reference-manual/js/

[33] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. 2012. Race Detection for Web Applications. *SIGPLAN Not.* 47, 6 (jun 2012), 251–262. https://doi.org/10.1145/2345156.2254095

[34] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2011. LOCKSMITH: Practical Static Race Detection for C. *ACM Trans. Program. Lang. Syst.* 33, 1, Article 3 (jan 2011), 55 pages. https://doi.org/10.1145/1889997.1890000

[35] Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective Race Detection for Event-Driven Programs. *SIGPLAN Not.* 48, 10 (oct 2013), 151–166. https://doi.org/10.1145/2544173.2509538

[36] Navid Salehnamadi, Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek. 2020. *ER Catcher: A Static Analysis Framework for Accurate and Scalable Event-Race Detection in Android.* Association for Computing Machinery, New York, NY, USA, 324–335. https://doi.org/10.1145/3324884.3416639

[37] sharat87. 2021. *Fix race condition in leaving rooms by sharat87.* Socket.IO. Retrieved May 4, 2022 from https://github.com/socketio/socket.io-adapter/pull/74

[38] Soulike. 2021. *Possible race condition on variable requestTemplate and roleTemplate.* Telefónica I+D. Retrieved January 18, 2023 from https://github.com/telefonicaid/fiware-pep-steelskin/issues/477

[39] Soulike. 2023. *NodeRT-OpenSource/NodeRT-OpenSource.* NodeRT-OpenSource. Retrieved January 18, 2023 from https://github.com/NodeRT-OpenSource/NodeRT-OpenSource

[40] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. 2018. Efficient Dynamic Analysis for Node.Js. In *Proceedings of the 27th International Conference on Compiler Construction* (Vienna, Austria) *(CC 2018)*. Association for Computing Machinery, New York, NY, USA, 196–206. https://doi.org/10.1145/3178372.3179527

[41] Haiyang Sun, Daniele Bonetta, Filippo Schiavio, and Walter Binder. 2019. Reasoning about the Node.Js Event Loop Using Async Graphs. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, Piscataway, NJ, 61–72. https://doi.org/10.1109/CGO.2019.8661173

[42] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. Detecting and Surviving Data Races Using Complementary Schedules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 369–384. https://doi.org/10.1145/2043556.2043590

[43] Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. 2017. A Comprehensive Study on Real World Concurrency Bugs in Node.Js. In

Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, Piscataway, NJ, 520–531. https://doi.org/10.1109/ASE.2017.8115663

[44] Weihang Wang, Yunhui Zheng, Peng Liu, Lei Xu, Xiangyu Zhang, and Patrick Eugster. 2016. ARROW: Automated Repair of Races on Client-Side Web Pages. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 201–212. https://doi.org/10.1145/2931037.2931052

[45] Diyu Wu, Dongjie He, Shiping Chen, and Jingling Xue. 2020. Exposing Android Event-Based Races by Selective Branch Instrumentation. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Press, Piscataway, NJ, 265–276. https://doi.org/10.1109/ISSRE5003.2020.00033

[46] Jingyue Wu, Heming Cui, and Junfeng Yang. 2010. Bypassing Races in Live Applications with Execution Filters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI’10)*. USENIX Association, Vancouver, BC, 135–149. https://www.usenix.org/conference/osdi10/bypassing-races-live-applications-execution-filters

[47] Tong Zhang, Changhee Jung, and Dongyoon Lee. 2017. ProRace: Practical Data Race Detection for Production Use. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi’an, China) *(ASPLOS ’17)*. Association for Computing Machinery, New York, NY, USA, 149–162. https://doi.org/10.1145/3037697.3037708

[48] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2016. TxRace: Efficient Data Race Detection Using Commodity Hardware Transactional Memory. *SIGARCH Comput. Archit. News* 44, 2 (mar 2016), 159–173. https://doi.org/10.1145/2980024.2872384

[49] Yunhui Zheng, Tao Bao, and Xiangyu Zhang. 2011. Statically Locating Web Application Bugs Caused by Asynchronous Calls. In *Proceedings of the 20th International Conference on World Wide Web* (Hyderabad, India) *(WWW ’11)*. Association for Computing Machinery, New York, NY, USA, 805–814. https://doi.org/10.1145/1963405.1963517

[50] Qing Zhou, Lian Li, Lei Wang, Jingling Xue, and Xiaobing Feng. 2018. May-Happen-in-Parallel Analysis with Static Vector Clocks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 228–240. https://doi.org/10.1145/3168813

Received 2023-02-16; accepted 2023-05-03