Race Detection for Event-Driven Node.js Applications

Xiaoning Chang[†‡], Wensheng Dou[†‡*¶], Jun Wei[†‡], Tao Huang[†‡¶], Jinhui Xie[§], Yuetang Deng[§], Jianbo Yang[§], Jiaheng Yang[§]

[†]*State Key Lab of Computer Sciences, Institute of Software, Chinese Academy of Sciences, Beijing, China*

[‡]*University of Chinese Academy of Sciences, Beijing, China*

[*]*Nanjing Institute of Software Technology, Nanjing, China*

[§]*Tencent, Inc., Guangzhou, China*

[†]{changxiaoning17, wsdou, wj, tao}@otcaix.iscas.ac.cn, [§]{hugoxie, yuetangdeng, xiaotuoyang, jiahengyang}@tencent.com

***Abstract*—Node.js has become a widely-used event-driven architecture for server-side and desktop applications. Node.js provides an effective asynchronous event-driven programming model, and supports asynchronous tasks and multi-priority event queues. Unexpected races among events and asynchronous tasks can cause severe consequences. Existing race detection approaches in Node.js applications mainly adopt random fuzzing technique, and can miss races due to large schedule space.**

In this paper, we propose a dynamic race detection approach *NRace* for Node.js applications. In NRace, we build precise happens-before relations among events and asynchronous tasks in Node.js applications, which also take multi-priority event queues into consideration. We further develop a predictive race detection technique based on these relations. We evaluate NRace on 10 real-world Node.js applications. The experimental result shows that NRace can precisely detect 6 races, and 5 of them have been confirmed by developers.

***Index Terms*—Node.js, event-driven architecture, race detection**

I. Introduction

Node.js is an increasingly popular event-driven architecture, and widely used in server-side and desktop applications. The official Node.js package manager *npm* [1] has become the largest package registry and consists of more than 1,500,000 building blocks in April 2021. Nowadays, 50% professional developers use Node.js to develop their frameworks, libraries and tools [2]. Industrial giants, such as PayPal [3], Uber [4] and Yahoo [5], also widely adopt Node.js in their systems.

Node.js adopts an event-driven architecture, and provides an effective asynchronous programming model. In Node.js, time-consuming IO operations, e.g., file access operations, can be delegated as asynchronous tasks, running in the dedicated threads in libuv [6]. Thus, Node.js applications are not blocked by these time-consuming IO operations. Once an asynchronous task completes, a completion event is put into certain event queue. Different from other event-driven architectures, e.g., client-side JavaScript [7] and Android [8], [9], Node.js provides multiple event queues, which have different priorities. These events in different event queues are scheduled by the looper thread, i.e., the main thread in Node.js, based on their priorities. Note that, asynchronous tasks are concurrently executed in the dedicated underlying threads, which are different from the looper thread.

The above asynchronous programming model in Node.js can introduce races. First, since asynchronous tasks and their corresponding events are executed asynchronously, unordered executions among events can cause unexpected interleavings by developers, thus taking the application into faulty states. Second, unordered asynchronous tasks and events can access to the same external resource e.g., files, thus introducing races. Races in Node.js applications can cause severe consequences, e.g., unexpected application states, and even worse system crashes [10], [11]. As server-side applications, races in Node.js applications may affect many end users. Thus, it is important for Node.js developers to automatically detect races in Node.js applications.

Existing approaches on race detection in event-driven architectures mainly focus on client-side JavaScript applications [7], [12]–[17] and Android applications [8], [9], [18]–[20]. Race detection approaches in client-side JavaScript applications [7], [12]–[17] mainly concern programming model features in browsers, e.g., DOM and AJAX. Race detection approaches in Android applications [8], [9], [18]–[20] mainly concern Android GUI model and the multi-thread programming model. Note that, all these approaches only consider *one event queue* while Node.js supports *multiple event queues* and schedules events with *different priorities*. Therefore, it is challenging to apply these approaches to Node.js applications, although programming models of browsers, Android and Node.js are conceptually similar. Recently, a few race detection approaches have been proposed for Node.js applications. Node.fz [11] adopts the fuzzing technique to randomly perturb event schedules. NodeRacer [21] further utilizes happens-before relations to eliminate infeasible event schedules. However, they only expose limited schedule space, and miss races. NodeAV [22] only detects atomicity violations in Node.js, and ignores other kinds of races.

In this paper, we propose *NRace*, a predictive race detector for Node.js applications. Given a Node.js application, NRace records its execution trace, and builds precise happens-before relations among asynchronous tasks and events. Further,

[¶]Wensheng Dou and Tao Huang are the corresponding authors.

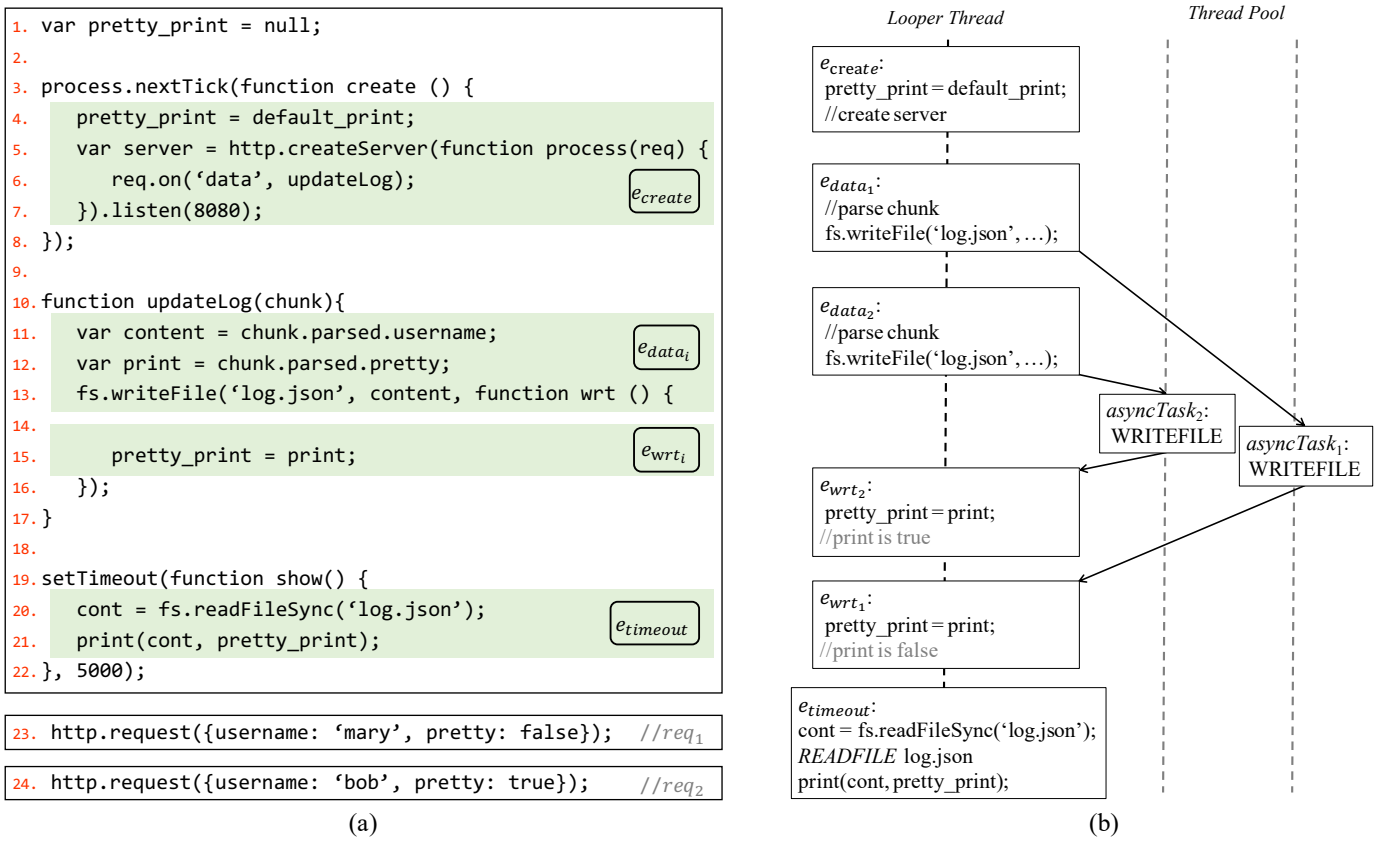

Fig. 1. A simplified Node.js application. (a) shows a simple web server and two requests. (b) shows its execution process, in which e_{data_i} and e_{wrt_i} represents the events for request req_i ($i \in \{1, 2\}$). We present actual file operations in capital letters, i.e., $WRITEFILE$ and $READFILE$, and distinguish them from related API invocations, e.g., $fs.writeFile$.

NRace predicts races in alternative executions that satisfy happens-before relations. NRace takes asynchronous tasks and multi-priority event queues into consideration, and proposes an efficient algorithm to build and query happens-before relations in real-world Node.js applications. To reduce benign races, we further propose a few commutative race patterns based on frequently observed benign races.

To demonstrate the effectiveness of NRace, we evaluate NRace on real-world Node.js applications from two aspects. First, we evaluate NRace for bug detection on 10 known races in real-world Node.js applications. The experimental result shows that NRace is able to effectively detect all 10 known races while the state-of-the-art fuzzing technique NodeRacer only detects 6 races in three hours. Second, we further apply NRace on 10 open-source Node.js applications, and detect 6 previously unknown races, 5 of which have been confirmed by developers. NRace and its experimental subjects are available at https://github.com/tcse-iscas/nrace.

We summarize our main contributions as follows:

- We propose a predictive race detector for Node.js applications, and build precise happens-before relations that can handle asynchronous tasks and multi-priority event queues in Node.js applications.
- We implement our approach as NRace and evaluate it on real-world Node.js applications. The experimental results show that NRace can effectively detect races in real-world Node.js applications.

II. Motivation

In this section, we present an illustrative example to explain the Node.js event-driven programming model and races rising in Node.js applications.

A. Motivating Example

Figure 1(a) shows a simplified Node.js web server application (Line 1-22) and two requests for it (Line 23-24).

In this web server application, it first uses API $process.nextTick()$ to register event e_{create} of type *NextTick* (Line 3). Then, it registers event $e_{timeout}$ of type *Timeout* to execute after 5000 milliseconds (Line 19). When processing e_{create}, the looper thread executes its callback $create()$ to write the default setting to variable $pretty\_print$ (Line 4) and to create a server (Line 5). The server registers event e_{data} to process user requests (Line 6). Once a request arrives, event e_{data} is triggered. The looper thread processes e_{data} and executes its callback $updateLog()$ with parameter $chunk$, which represents the received data of request. Callback

$updateLog()$ parses $chunk$ (Line 11-12) and logs username into log file $log.json$ (Line 13).

In order not to block the looper thread, callback $updateLog()$ delegates the time-consuming file writing operation as an asynchronous task $asyncTask$ to the thread pool and registers event e_{wrt} to execute after the completion of $asyncTask$ (Line 13). While the looper thread proceeds, a thread in the thread pool asynchronously writes file $log.json$. On the completion of asynchronous task $asyncTask$, event e_{wrt} is triggered. The looper thread executes its callback $wrt()$ to update the variable $pretty\_print$ according to the request (Line 15).

For each request, two events e_{data} and e_{wrt}, and an asynchronous task $asyncTask$ are executed. Let's assume that two requests req_1 and req_2 with different pretty-print configurations simultaneously arrive (Line 23-24). Event e_{data} and e_{wrt} and asynchronous task $asyncTask$ will be executed twice, once per request. For simplicity, we denote e_{data_i}, e_{wrt_i} and $asyncTask_i$ for each request req_i ($i \in \{1, 2\}$).

Once 5000 milliseconds elapse, event $e_{timeout}$ is triggered. The looper thread executes its callback $show()$ to synchronously read log file $log.json$ (Line 20) and print content $cont$ with the pretty-print setting $pretty\_print$ (Line 21).

Note that, in Node.js, once an event is generated, it is put into an event queue. Node.js maintains several event queues to hold different types of events, i.e., *NextTick*, *Promise*, *Immediate*, *Timeout* and *IO* [23]. Events in the same event queue are scheduled in the order that they are enqueued.

The looper thread processes *Immediate*, *Timeout* and *IO* event queues in a round-robin manner. When an event queue is exhausted or the number of processed events reaches a threshold, the looper thread switches to process the next event queue. However, if there are events of *NextTick* type, these *NextTick* events will be processed first before the loop thread processes any event [23]. Therefore, in Figure 1(a), e_{create} is processed before $e_{timeout}$.

B. Races in Node.js Applications

For each request in Figure 1(a), two events e_{data} and e_{wrt}, and an asynchronous task $asyncTask$ are executed in order $e_{data} \rightarrow asyncTask \rightarrow e_{wrt}$. However, two executions of req_1 and req_2 can interleave and thus cause races. Figure 1(b) shows an execution of races for two requests. We can see three races in Figure 1(b).

(1) Event $e_{timeout}$ can be triggered before $asyncTask_1$'s completion event e_{wrt_1}. Consequently, event $e_{timeout}$ pretty prints the content of $log.json$. Compared with the execution shown in Figure 1(b), this is a race between event $e_{timeout}$ and e_{wrt_1} on variable $pretty\_print$, causing the print format to be non-deterministic.

(2) Asynchronous task $asyncTask_1$ and $asyncTask_2$ are concurrently executed in the thread pool and their processing order in the file system is unknown. Thus, the content of file $log.json$ is non-deterministic. This is a race between $asyncTask_1$ and $asyncTask_2$ on file $log.json$.

(3) Event $e_{timeout}$ and asynchronous task $asyncTask_1$ can run concurrently. Their execution order in the file system is uncertain. Therefore, this is a race between event $e_{timeout}$ and asynchronous task $asyncTask_1$ on file $log.json$.

C. Approach Overview

In order to detect races in Node.js applications, we need to address two technical challenges. First, in addition to memory locations, how can we model accesses on external resources, e.g., file $log.json$ in Figure 1(a)? As shown in the above example, external resources are contended by events and asynchronous tasks. In Node.js, external resources are managed by underlying system and opaque to developers. Second, Node.js has its special execution mechanism for events and asynchronous tasks, e.g., the execution order between e_{create} and $e_{timeout}$ caused by the multi-priority event queues. How can we design precise happens-before relations among events and asynchronous tasks?

For the first challenge, we study the file system APIs in Node.js and model them into several basic file access operations with various types, e.g., *Crate*, *Read*, *Write* and *Delete*. Then, we further model these file access operations according to whether they are synchronous. Finally, we build precise conflicting patterns on these file access operations. Therefore, we can detect races among events and asynchronous tasks on external resources. Second, we build precise happens-before relations among events and asynchronous tasks. Specially, we design an efficient algorithm to build happens-before relations caused by multi-priority event queues.

III. Approach

Figure 2 presents the overview of NRace. Given a Node.js application with its test cases, NRace can predictively detect potential races. First, we run the Node.js application to profile its execution trace (Section III-A). Then, we design happens-before relations for Node.js applications, which can reflect the partial order among events and asynchronous tasks (Section III-B). We further design an efficient algorithm to build the happens-before graph for the collect execution trace. (Section III-C). Finally, we detect potential races on conflicting operations that are not ordered by the happens-before graph (Section III-D) and filter out benign races using predefined commutative race patterns (Section III-E).

A. Trace Collection

An execution trace of a Node.js application is a sequence of operations, which are performed by events or asynchronous tasks. In the following paper, we uniformly call events and asynchronous tasks as *actions* for simplicity, when we need to unify events and asynchronous tasks.

Lifecycle related operations. Lifecycle related operations are used to control the generation and execution of events and asynchronous tasks. We summarize them as follows.

- $start(a)$: start executing action a, i.e., an event or an asynchronous task.

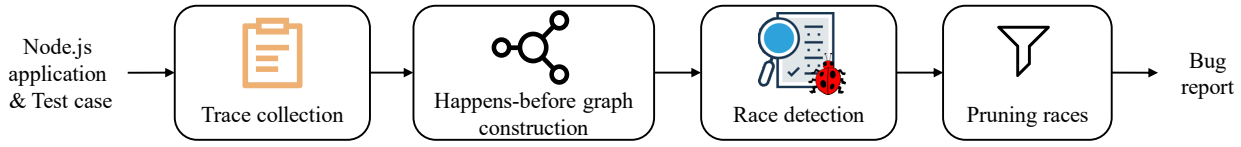

Fig. 2. The overview of NRace.

- $end(a)$: end executing action a, i.e., an event or an asynchronous task.
- $delegate(e, t)$: event e delegates asynchronous task t to the thread pool.
- $register(e_i, e_j)$: event e_i registers event e_j. There are two cases for event registration. First, event e_i registers event e_j to execute after the completion of an asynchronous task. For example, e_{data_1} registers e_{wrt_1} for $asyncTask_1$ in Figure 1. Second, Node.js allows developers to schedule events with several built-in APIs, e.g., $process.nextTick()$, $setImmediate()$, $setInterval()$ and $setTimeout()$. That said, event e_j is registered by event e_i and will be executed after a period of time. Events registered by API $process.nextTick()$, $setImmediate()$, $setInterval()$ and $setTimeout()$ are of type of *NextTick*, *Immediate* and *Timeout*[1], respectively.
- $trigger(t, e)$: event e is triggered by the completion of asynchronous task t and put into the corresponding event queue.
- $trigger(e_i, e_j)$: event e_j is triggered by event e_i and put into the corresponding event queue.
- $resolve(e, p)$: When promise p is created, a resolved event $p.resolved$ is registered to be executed once the promise p is resolved (fullfilled) [24]. $resolve(e, p)$ denotes that event e resolves promise p.
- $p_{all} = Promise.all([p_1, p_2, ..., p_n])$: API $Promise.all()$ returns promise p_{all}, which will be resolved after all promises $p_1, p_2, ..., p_n$ are resolved, or rejected after one of promises $p_1, p_2, ..., p_n$ is rejected.
- $p_{race} = Promise.race([p_1, p_2, ..., p_n])$: API $Promise.race()$ returns promise p_{race}, which will be resolved or rejected after one of promises $p_1, p_2, ..., p_n$ is resolved or rejected.

Resource access operations. Resource access operations include memory access operations and external resource access operations. We summarize them as follows.

- $read(v, val, e)$: event e reads memory location v and obtains value val. We consider both reading variables and getting fields of objects as $read$ operations.
- $write(v, val, e)$: event e writes value val to memory location v. Similarly, we consider writing variables, putting fields and deleting fields of objects as $write$ operations.
- $fileAccess(f, type, a)$: acton a, i.e., an event or an asynchronous task, accesses file f with access type $type$. To model external resource accesses, we map each file system API into one or multiple operations. In particular, we support seven access types on files, i.e., *Create*, *Open*, *Read*, *Write Close*, *Delete*, and *Stat*. For example, $fs.readFileSync(bar.txt)$ invoked by event e is modeled as $fileAccess(bar.txt, Read, e)$. We model an asynchronous file access into a sequence of lifecycle related operations and file access operations. For example, $fs.readFile(bar.txt, cb)$ invoked by event e is modeled as a sequence of operations: $delegate(e, t)$, $start(t)$, $fileAccess(bar.txt, Read, t)$, $end(t)$ and $start(cb)$, where t is the asynchronous task that performs the file access operation and cb is the completion event of asynchronous task t.

Other operations. In addition to above operations, we also track following operations, which are used to filter benign races in Section III-E.

- $conditional(val, e)$: event e performs a $if - check$ and the checking result is val.
- $binary(opt, left, right, val, e)$: event e performs a binary operation, whose left and right operands are $left$ and $right$, respectively. The operator and result of binary operation is opt and val, respectively.

In NRace, we utilize *async_hooks* [25] to track lifecycle related operations. async_hooks provides several functions to record lifecycle related operations, e.g., $init()$, $before()$, $after()$, $promiseResolve()$. For example, every time an event e is registered, function $init()$ is called to track the type of event e and the event that registers e. We further utilize Jalangi [26] to track the remaining operations. For example, Jalangi records the names of variables and the values written into the variables for $write$ operations so that it can track information for resource access operations.

B. Happens-Before Relation

Given an execution trace τ, happens-before relation $\prec$ is a partial relation among actions in trace τ, i.e., events and asynchronous tasks. For action act_i and act_j, we denote act_i happens before act_j as $act_i \prec act_j$.

In addition, we also build happens-before relations over operations performed by actions. For operation op_i and op_j, we denote op_i happens before op_j as $op_i \prec op_j$. We overload happens-before operator $\prec$ for both actions and operations.

Rule 1 (Transitivity): The happens-before relation $\prec$ is transitive, i.e., for action act_i, act_j and act_k, if $act_i \prec act_k$ and $act_k \prec act_j$, then $act_i \prec act_j$.

Rule 2 (Program-Order): Operations performed by the same action are deterministically executed in the program order. If operation op_i and op_j are executed by the same action and op_i occurs before op_j in trace τ, then $op_i \prec op_j$.

[1]Events registered by $setInterval()$ and $setTimeout()$ are of type *Timeout*.

Rule 3 (Event-Atomicity): Each event is executed without interruption. That said, for operation op_i and op_j performed by event e_i and e_j respectively, if $op_i \prec op_j$, then any operation in e_i happens before any operation in e_j.

Rule 4 (Event-Registration): Each event needs to be registered before it is processed. That said, if event e_j is registered by event e_i, then $e_i \prec e_j$.

Rule 5 (SetInterval): If events $e_1, e_2, ..., e_n$ are registered in order via API $setInterval()$, then these events are executed in the registration order, i.e., $e_i \prec e_{i+1}$ for $1 \leq i \leq n-1$.

Rule 6 (Promise-Resolve): If event e_i resolves promise p and event e_j is the resolved event associated with promise p, then $e_i \prec e_j$.

Rule 7 (Promise-All): If promises $p_1, p_2, ..., p_n$ are arguments passed to API $Promise.all()$ and promise p_i is resolved by event e_i, then p_{all} is resolved after $e_1, e_2, ..., e_n$. In other words, the resolved event e_{all} associated with p_{all} is executed after $e_1, e_2, ..., e_n$. Therefore, we build happens-before relation $e_i \prec e_{all}$, where $1 \leq i \leq n$.

Rule 8 (Promise-Race): If promises $p_1, p_2, ..., p_n$ are arguments passed to API $Promise.race()$ and promise p_i is resolved by event e_i, then p_{race} is resolved after one of $e_1, e_2, ..., e_n$. In other words, the resolved event e_{race} associated with p_{race} is executed after one of $e_1, e_2, ..., e_n$.

We build the happens-before relation among $e_1, e_2, ..., e_n$ and e_{race} in an alternative manner. We first check happens-before relations among $e_1, e_2, ..., e_n$. If there exists two events e_i and e_j such that $e_i \prec e_j$, then we remove the corresponding promise p_j from arguments of API $Promise.race()$, where $1 \leq i, j \leq n$. If there is only one remaining promise p_k as the argument of API $Promise.race()$, we build happens-before relation $e_k \prec e_{race}$.

Rule 9 (AsyncTask-Delegation): If event e delegates an asynchronous task t, then $delegate(e, t) \prec start(t)$.

Rule 10 (AsyncTask-Completion): If the completion of asynchronous task t triggers event e, then $t \prec e$.

Rule 11 (FIFO): Events of the same type are put into the same event queue. The looper thread processes events in the same event queue in FIFO order. For event e_i and e_j with the same type, i.e., $NextTick$, $Immediate$ and $Promise$, if e_i is registered before e_j is registered, then $e_i \prec e_j$. In particular, for IO event e_i and e_j, if e_i is triggered before e_j, then $e_i \prec e_j$.

Rule 12 (FIFO-Timeout): Events of *Timeout* type are also processed in the FIFO manner. For event e_i and e_j of *Timeout* type, if e_i is registered before e_j is registered and the delay time of e_i is no more than that of e_j, then $e_i \prec e_j$.

Rule 13 (NextTick): Events of *NextTick* type hold the highest priority to be executed. For event e_i and e_j, where e_i is of *NextTick* type while e_j is other types, if e_i is registered before e_j is executed, then $e_i \prec e_j$.

Discussion. Some existing works, e.g., NodeRacer [21], NodeAV [22], and AsyncG [27], also propose some happens-before relations for Node.js applications. NodeRacer [22] and AsyncG [27] only focus on happens-before relations among events, and ignore happens-before relations between events and asynchronous tasks. They also ignore happens-before relations among operations. Thus, they do not contain happens-before rule 2, 3, 9 and 10. NodeAV [22] treats asynchronous IO tasks as synchronous operations, and cannot reflect happens-before rule 9 and 10. Further, NodeAV does not support happens-before rule 6, 7, 8, and ignores the delay time in rule 12.

Therefore, these existing works lack some key happens-before rules for detecting races in Node.js applications. Our happens-before relations can reflect the relations among events and asynchronous tasks, and relations among operations. Thus, our happens-before relations are more complete and precise than existing works. We believe that our happens-before relations can also benefit existing works.

C. Happens-Before Graph Construction

Based on an execution trace τ, we build a happens-before graph $G(V, E)$, in which node $v \in V$ is an action and edge $e \in E$ represents the happens-before relation among actions.

NodeRacer [21] and NodeAV [22] adopts the following algorithm to construct the happen-before graph. (a) The algorithm first adds all actions into the graph. (b) Then, it builds happens-before relations for simple rule 4-7 and rule 9-10, which do not depend on any other relations. (c) Next, it builds happens-before relations introduced by complex rules, i.e., Promise-Race, FIFO, FIFO-Timeout and NextTick, which depend on other relations. The algorithm checks whether each pair of actions satisfies one of complex rules. If yes, the corresponding relation is added into the graph.

Note that, in step (c), the newly added relations may introduce other happens-before relations on actions that we have evaluated through complex rules. Therefore, if step (c) finds a new relation, the graph needs to be reprocessed again. In other words, the graph is processed until no more relation is found. The above recursive process is time-consuming if there are many actions that can be applied for Promise-Race, FIFO, FIFO-Timeout and NextTick rules in trace τ.

In order to efficiently build the happens-before graph, we design an algorithm scalable to real-world Node.js applications, as shown in Algorithm 1. Our algorithm reduces the overhead of happens-before graph construction from following aspects:

- *Incremental graph construction*: Our algorithm starts with empty set of nodes and edges (Line 1-2), and incrementally builds the happens-before graph in the trace order (Line 3). After action act is added into graph (Line 5), we only need to build happens-before relations on a *limited* number of actions in the graph (Line 6-7), instead of *all* of actions in NodeRacer [21] and NodeAV [22].
- *Efficient rule matching*: Given an action act, we efficiently find action act' that happens before act for both simple and complex rules (Line 6-7). In particular, it optimizes the happens-before relation construction for complex rules so that it does not perform the recursive process and reduces the overhead, which is done via function $buildComplexHB()$ (Line 7).

Algorithm 1: Happens-before graph construction

Input: τ (Execution trace)
Output: $G(V, E)$ (Happens-before graph)

```
1  V ← ∅;
2  E ← ∅;
3  for i ← 1; i ≤ τ.length; i++ do
4  |   act ← τ[i];
5  |   V ← V ∪ {act};
6  |   buildSimpleHB(V, E, act);
7  |   buildComplexHB(V, E, act);
8  end
9  Function buildComplexHB(V, E, act)
10 |   applyRaceRule(act);
11 |   U ← selectUnorderedAction(act);
12 |   U' ← sortUnorderedAction(U);
13 |   for j ← 1; j ≤ U'.length; j++ do
14 |   |   act' ← U'[j];
15 |   |   for rule ∈ complexRules do
16 |   |   |   if isMatch(act', act, rule) then
17 |   |   |   |   E ← E ∪ {(act', act)};
18 |   |   end
19 |   end
20 end
```

- *Efficient reachability query*: Since there is a large number of actions in the happens-before graph, it is time-consuming to perform the breadth-first graph search to query reachability. We improve the breadth-first search by stopping exploring impossible paths in advance.

We illustrate the above three ingredients as follows.

Incremental graph construction. We observe that, given an action act to be added into the graph, only actions that occur before act in trace τ may happen before act. Otherwise, trace τ will be infeasible. Based on this observation, we incrementally build the happens-before graph, by adding one action act into the graph according to the trace order (Line 3). For each added action act (Line 5), we only need to build happens-before relations between action act and other actions in the graph.

Note that, event e_{race} involved in Promise-Race rule is processed differently because the relation caused by Promise-Race rule on e_{race} is determined after $e_1, e_2, ..., e_n$ that are related to arguments of API $Promise.race()$ are added into the graph. Therefore, after all of $e_1, e_2, ..., e_n$ are added into graph, we add e_{race} into the graph and build happens-before relations for it.

Efficient rule matching. Given action act and rule $rule$, we find action act' that happens before action act and build happens-before relation between act' and act.

For simple rules, we directly find action act' and build happens-before relation between act' and act in a constant time. For example, when processing operation $register(e_1, e_2)$, we store registrar information e_1 in e_2. When evaluating registration rule for e_2, we find e_1 and obtain $e_1 \prec e_2$ directly.

We build happens-before relations for AsyncTask-Delegation rule in a special manner. For this rule, we cannot obtain $delegate(e, a) \prec start(a)$, because there are only actions but not operations in the happens-before graph. In order to build relations between e and a, we make a compromise: NRace builds the edge from e to a in the happens-before graph. We will handle this case when querying happens-before relations among operations, and it causes no false happens-before relation among operations. Thus, our race detection cannot be compromised.

For complex rules, we first evaluate Promise-Race rule for act (Line 10), since the Promise-Race rule only depends on the relation on the set of events corresponding to arguments of API $Promise.race()$, which has been determined.

Then, we evaluate the remaining complex rules, i.e., FIFO, FIFO-Timeout and NextTick rule for act (Line 11-19). In order to avoid the recursive process, our algorithm performs following optimizations.

Optimization 1: Find unordered actions. As discussed earlier, NodeRacer attempts to build relations for complex rules on each pair of actions in step (c), and wastes time on evaluating pairs of actions that have already been ordered by happens-before relations. This motivates us to avoid applying complex rules on ordered actions.

We adopt chain decomposition [9], [13] to find unordered actions. The idea of chain decomposition is to assign actions to chains so that actions on the same chain are ordered by happens-before relations, and actions on the different chains may be unordered. We use $a \triangleright c$ to denote that action a is on chain c.

Our chain decomposition algorithm is described as follows. When we add action act into the happens-before graph by simple rules, we greedily assign action act to a chain. (i) We first find the set of actions A, where actions in A have happens-before edges with act. (ii) If there exists an action $act' \in A$, after adding act into act''s chain c', chain c' does not diverge, then we assign act to chain c'. (iii) If such an action does not exist, we create a new chain and assign act to it. For example, in Figure 3, before adding e_7 into the graph, $e_1, e_2, ..., e_6$ are assigned to two chains c_1 and c_2. When adding e_7 into the graph, we find $e_6 \prec e_7$, and assigning e_7 to chain c_2 does not make c_2 diverge. Therefore, e_7 is added into chain c_2.

Given a newly added action act, function $selectUnorderedAction()$ (Line 11) finds actions, which are unordered with act. Let c_{act} denote the chain that act belongs to, i.e., $act \triangleright c_{act}$. For each chain c that is different from c_{act}, we identify its unordered actions with act through the following process. We find the last action act_i in c, which satisfies the following condition: $\exists act_j \triangleright c_{act}$, $act_i \prec act_j \& (act_j \prec act || act_j = act)$. Actions that happen after act_i in chain c are unordered with act. For example, in Figure 3, when action e_7 is added into the graph, we find that e_6 happens before e_7 on chain c_2, and e_2 on chain c_1 happens before e_6. Therefore, e_3 and e_4 on chain c_1 are unordered with e_7.

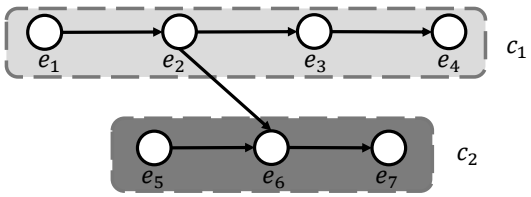

Fig. 3. A happens-before graph example with chain decomposition. Two chains c_1 and c_2 are denoted by light grey and dark grey, respectively.

Optimization 2: Determine action evaluation order. As discussed earlier, NodeRacer recursively applies complex rules until the graph reaches a fixpoint. In order to avoid the recursive process, we determine an evaluation order for unordered actions via function $sortUnorderedAction()$ (Line 12). For action $act_a, act_b \in U$, if act_a happens before act_b on the same chain, act_a is sorted before act_b in U'. Then, we apply complex rules between action $act' \in U'$ and act (Line 13-19).

Note that, the newly added relation on act' and act only introduces relations on actions that are registered by act'. For example, in Figure 3, if we add the relation from e_3 to e_7, denoted as (e_3, e_7), which is not present in the graph, relation (e_3, e_7) only introduces relations on actions registered by e_3, e.g., e_4. These successor actions happen after act' and are not evaluated. Therefore, if we evaluate actions following the chain order, we do not need to recursively evaluate complex rules on the graph.

Efficient reachability query. The reachability among nodes in the happens-before graph G reflects happens-before relations among events and asynchronous tasks. For node u and v, iff there is a path from u to v in G, $u \prec v$.

We adopt breadth-first graph search to query reachability in the happens-before graph G. The breadth-first graph search has a maximum time complexity of $O(E)$. However, since Node.js applications generates a large number of actions in a short time, there is a large number of nodes along with edges in the happens-before graph. The breadth-first graph search does not scale to real-world Node.js applications.

We observe that, if node n_k does not occur before node n_j in the trace τ, denoted as $n_k \not\prec_\tau n_j$, then $n_k \not\prec n_j$ and there is no path from n_i to n_j through n_k. Based on this observation, to speed up querying reachability from node n_i to node n_j, we stop traversing to node n_k, where $n_k \not\prec_\tau n_j$. The *benefit* is that, we stop exploring impossible paths in advance, thus improving efficiency.

Based on happens-before relations among actions, we can determine happens-before relations among operations. For operation op_i and op_j, $op_i \prec op_j$ if:

- op_i and op_j are performed by the same action and $op_i \prec_\tau op_j$ or
- op_i and op_j are performed by different actions and $action(op_i) \prec action(op_j)$, where $action(op)$ denotes the action that performs operation op.

When querying reachability, we deal with the compromise case introduced by AsyncTask-Delegation rule when building the happens-before graph. If $action(op_i)$ delegates $action(op_j)$ and the delegation operation occurs before op_i in $action(op_i)$, then $op_i \not\prec op_j$.

We define concurrency relation as $Con(op_i, op_j) = op_i \not\prec op_j \wedge op_j \not\prec op_i$. That said, if neither op_i happens before op_j nor op_j happens before op_i, op_i has the concurrency relation with op_j. We can further refine the above concurrency relation as $Con(op_i, op_j) = op_i \prec_\tau op_j \wedge op_i \not\prec op_j \vee op_j \prec_\tau op_i \wedge op_j \not\prec op_i$, to reduce the number of reachability queries.

TABLE I
FILE ACCESS CONFLICTING PATTERNS

	Create	Delete	Read	Write	Open	Close	Stat
Create	◇	◇	◇	◇	◇	◇	◇
Delete	◇	◇	◇	◇	◇	◇	◇
Read	◇	◇		◇	◇	◇	
Write	◇	◇	◇	◇	◇	◇	
Open	◇	◇	◇	◇		◇	
Close	◇	◇	◇	◇	◇	◇	
Stat	◇	◇					

If two types intersect at ◇, they form a conflicting pattern.

D. Race Detection

In Node.js applications, conflicting operations can be operations that access to memory locations and external resources, e.g., files. We explain them as follows.

- Conflicting memory access operations. Two memory access operations conflict when they access the same memory location and at least one of them is $write$ operation.
- Conflicting file access operations. We define file conflicting patterns based on the equivalent influence of access type on file system. The file conflicting patterns are shown in Table I. Two file access operations conflict when they access the same file and their access type match one of our predefined conflicting patterns. Note that, our pattern is more powerful than NodeAV [22], in which file access operations are only modeled as *Read* and *Write*. For example, $fs.statSync()$ is modeled as an operation of *Read*, and $fs.writeFileSync()$ is modeled as an operation of *Write*. However, they do not conflict in fact. Our file conflicting patterns can precisely describe this un-conflicting case.

Consider two operations op_i and op_j, a race exists between op_i and op_j, denoted as $Race(op_i, op_j)$, if (1) they conflict and (2) op_i have concurrency relation with op_j, i.e., $Con(op_i, op_j) = true$.

We first process the observed trace τ to obtain the set of operations that access the same resource x: $Op(x) = \{op_i | op_i \text{ accesses resource x}\}$. For each pair of operations (op_i, op_j) in $Op(x)$, we check whether they have concurrency relation, i.e., $Con(op_i, op_j) = true$.

E. Pruning Races

We observe that the above detection approach by finding unordered conflicting operations leads to many benign races. We find that some commutative operations do not need to be ordered by happens-before relations to ensure correctness. To

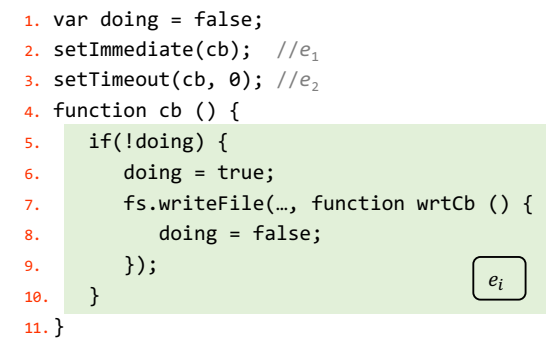

Fig. 4. Commutative pattern summarized from ad-hoc synchronization, where e_i represents event e_1 and e_2.

address this issue, we summarize three commutative patterns to automatically filter benign races.

Sequentialize actions. Node.js developers often utilize ad-hoc synchronization to force multiple actions to execute in sequence. For instance, in Figure 4, event e_1 and e_2 is registered by API $setImmediate()$ and $setTimeout()$, respectively (Line 2-3). Event e_1 and e_2 delegates asynchronous file writing task $asyncTask_1$ and $asyncTask_2$, respectively (Line 7). If one event (e.g., e_1) launches an asynchronous task (e.g., $asyncTask_1$), variable $doing$ is set to $true$. If another event (e.g., e_2) is executed before the previous asynchronous task (e.g., $asyncTask_1$) completes, it does not launch its asynchronous task (e.g., $asyncTask_2$). In this way, developers prevent multiple asynchronous tasks from being concurrently executed. Therefore, although event e_1 has a race with event e_2 on variable $doing$, there is no harmful impact on the application.

To detect this benign race, we analyze *conditional* and *write* operations to identify the variable that sequentializes actions. If a variable v is read in the *conditional* operation and two *write* operations on variable v happen before and after an action, respectively, then variable v protects actions from races. Any race on this variable is regarded as a benign race.

In our approach, we utilize Jalangi [26] to record $conditional$ operations. Since Jalangi only records values but not variables for $conditional$ operations, we utilize $read(x_r, val_r, e)$ operation that occurs before $conditional(val_c, e)$ to infer the variable accessed by the $conditional$ operation. If $val_r = val_c$, then variable v_r accessed by $read$ operation is regarded as the variable accessed by $conditional$ operation.

Use counters. Node.js applications often utilize counters. One case is that multiple events increase a counter. No matter which order these events are executed in, the counter is correctly increased. Another case is that, after a Node.js applications increases a counter, it checks whether the value of the counter equals to a given threshold. If true, the program will execute some functions. These $read$ and $write$ operations on the counter have no harmful impact on the application, since the counter does not reach the threshold.

A variable v is a counter variable if it satisfies the following conditions. (a) Variable v is used by three operations in sequence, i.e., $read$, $binary$ and $write$. (b) Variable v is read by a $read$ operation, and written by a $write$ operation. (c) $binary$ operation's operator is addition or subtraction (i.e., $+$ or $-$), and its two operands are of type $Number$. (d) $binary$ operation's left operand (or right operand) uses variable v, and its return result uses variable v.

Similar to $conditional$ operations, Jalangi only records values but not variables for $binary$ operations. We utilize $read(x_r, val_r, e)$ and $write(x_w, val_w, e)$ operation that occur before and after $binary(op, left, right, val_b, e)$ to infer the variable accessed by $binary$ operation. If $var_r = left$ (or $var_r = right$), we treat variable v_r is used as left (or right) operand of $binary$ operation. If $var_b = val_w$ we treat variable x_w is used as return result of $binary$ operation.

Write shared resource with the same value. Some operations are commutative because they write the shared resource with the same value. The shared resource holds the same value no matter in which order two events are executed. Therefore, for two operations $write_1(x, val_1, e_1)$ and $write_2(x, val_2, e_2)$, if they are unordered and val_1 equals to val_2, we regard them as commutative operations.

Note that, the commutative patterns we design are not sound. Without mining developer intention and semantics of operations, we cannot ensure the commutativity between actions and operations. For example, counter variables identified by our patterns are only self-increasing counters and actual counters are missed. However, we manually inspect benign races reported in our evaluation and find our patterns can identify more than half of benign races.

IV. Evaluation

Our evaluation answers the following research questions:

- **RQ1**: Can NRace detect known races in real-world Node.js applications? How does NRace compare with the state-of-the-art fuzzer NodeRacer [21]?
- **RQ2**: Can NRace detect previously-unknown races in real-world Node.js applications?
- **RQ3**: What is the runtime overhead of NRace, compared with NodeRacer [21]?

A. Experimental Setup

Dataset-1: Known races. To evaluate whether NRace can detect known races in real-world Node.js applications, we collect 10 known races from NodeRacer [21]. NodeRacer provides 11 known races in Node.js applications along with their test cases. Since one of these 11 Node.js applications, *linter-stylint*, cannot be profiled via *async_hooks* module, we only perform the evaluation on 10 Node.js applications, as shown in Table II. Column *Project* refers to applications, column *Description* gives a brief description of each application, column *Issue ID* shows the issue report ID in GitHub, and column *Category* denotes locations where races happen. *Event* and *Async* denotes races happen on events and asynchronous tasks, respectively.

Dataset-2: Unknown races. To evaluate whether NRace can detect previously-unknown races in real-world Node.js

TABLE II
DATASET-1: KNOWN RACES IN REAL-WORLD NODE.JS APPLICATIONS

ID	Project	Description	Issue ID	Category
1	agentkeepalive	Support keepalive http agent	23	Event
2	fiware-pep-steelskin	TID's implementation of FIWARE PEP GE	269	Event
3	ghost	The headless CMS for publication	1,834	Event
4	node-mkdirp	Like mkdir-p, but in Node.js	2	Async
5	nes	A webSocket adapter plugin	18	Event
6	node-logger-file	File endpoint for cinovo-logger	1	Async
7	socket.io	Real-time application framework	1,862	Event
8	del	Delete files and directories	43	Async
9	simplecrawler	Flexible event driven crawler for Node.js	298	Event
10	xlsx-extract	Extract data from XLSX files	7	Async

TABLE III
DATASET-2: REAL-WORLD NODE.JS APPLICATIONS

ID	Project	Description	#Star	LoC
1	nedb	A JavaScript Database	12,200	1,531
2	node-http-proxy	A full-featured http proxy	11,800	502
3	baobab	JavaScript persistent data tree	3,100	198
4	simplecrawler	Flexible event driven crawler for Node.js	2,100	1,606
5	serve-static	Serve static files	1,200	159
6	nodejs-websocket	A websocket server and client module	646	689
7	ncp	Asynchronous recursive file copying	601	231
8	line-reader	Asynchronous line-by-line file reader	427	256
9	json-file-store	A simple JSON store	184	1,192
10	fiware-pep-steelskin	TID's implementation of FIWARE PEP GE	11	1,735

applications, we collect 10 Node.js applications from GitHub that satisfy following conditions: (1) The application is able to run on Node.js 8.6 or above, which supports the *async_hooks* module to track events. (2) The application offers available test cases so that we can use them to drive the application to collect execution trace. Finally, we collect 10 Node.js applications, as shown in Table III. Column *Project* refers to the application, column *Description* gives a brief description of the application, column *#Star* shows the number of stargazers on GitHub, and column *LoC* presents lines of JavaScript code, computed by tool *cloc* [28]. We can see that most of experimental applications are popular.

We conduct our experiments in following steps. First, we utilize NRace to instrument the source code of the target application and utilize the test case to drive the application to collect execution trace. Second, we run NRace to analyze the observed trace. To answer RQ1, we use NRace and NodeRacer to detect known races on the dataset-1. To answer RQ2, we perform NRace on dataset-2 to detect previously-unknown races. For both RQ1 and RQ2, we manually inspect the code to validate whether each detected race is real. To answer RQ3, we measure the runtime overhead of NRace and NodeRacer. In particular, in order to compare the overhead of happens-before graph construction between NRace and NodeRacer [21], we implemented the happens-before graph construction algorithm adopted by NodeRacer, because the data structure of trace and happens-before graph of NodeRacer is different from ours and cannot be integrated into NRace. We compare the runtime overhead of our proposed happens-before construction algorithm and NodeRacer's algorithm. All experiments were conducted on 4-core 2.4GHz with 16GB memory, running macOS Catalina release 10.15.6.

TABLE IV
DETECTION RESULT ON DATASET-1

ID	NRace					NodeRacer
	Detected	#Total	#HR	#BR	#FP	Detected
1	Yes	4	2	2 (2)	0	No
2	Yes	13	2	10 (7)	1	Yes
3	Yes	1	1	0 (0)	0	Yes
4	Yes	1	1	0 (0)	0	Yes
5	Yes	2	1	1 (1)	0	Yes
6	Yes	9	8	1 (0)	0	No
7	Yes	1	1	0 (0)	0	No
8	Yes	7	3	4 (4)	0	Yes
9	Yes	11	1	7 (4)	3	Yes
10	Yes	3	1	0 (0)	2	No
Total	10	52	21	25 (18)	6	6

B. Detect Known Races

We run NRace on dataset-1 to measure the NRace's ability to detect known races. The result is shown in Table IV. Columns 2-6 present the result of NRace, where column *Detected* indicates whether the known race is detected by NRace. Column *#Total* presents the total number of races that NRace reports, calculated by $\#HR + \#BR + \#FP$, where *#HR*, *#BR* and *#FP* represents the number of harmful races, benign races and false positives reported by NRace, respectively. Note that the known race is included in *#HR* and the number of benign races automatically identified by NRace is shown in the bracket in column *#BR*.

As shown in Table IV, NRace detects all 10 known races. We further report 42 new races from 7 Node.js applications. After manual inspection, we find these new races can be classified into three categories: harmful races, benign races and false positives.

All of newly detected harmful races are side-effect races of the known races. For example, in project *fiware-pep-steelskin*, once a request req returns, event e_{req} writes variable $currentToken$ with returned data. However, if there are two requests req_1 and req_2, the execution order between event e_{req_1} and e_{req_2} is non-deterministic. If event e_{req_2} is executed before event e_{req_1}, e_{req_1} overwrites variable $currentToken$ with returned data of req_1, making request req_2 hangs. One of side-effect races is that, if event e_{req_1} throws an error after e_{req_2}, it also overwrites $currentToken$ with $null$, making req_2 hangs.

As shown in Table IV, 25 out of newly detected races are benign races, which is consistent with the fact that there is a large number of benign races in Node.js applications. Most of these benign races (18/25) are automatically identified by NRace.

False positives are determined by manual reproduction or related code review. As shown in Table IV, NRace reports 6 false positives. These false positives are caused by ad-hoc synchronization. For example, in project *simplecrawler*, event e_{add} caches a URL for crawling in future and event e_{req} invokes a request to crawl a given URL. If event e_{req} is executed before e_{add}, event e_{req} will find that there is no URL under crawling. Thus, it will invoke $stop()$ method to end up crawling. As a result, event e_{add} will not occur any more.

Note that, although we have detected all known races in our experiment, as a dynamic approach, we can still miss races if the provided test cases do not cover race-related events.

Comparison with NodeRacer. In order to compare NRace with the state-of-the-art fuzzer NodeRacer, we utilize NodeRacer to detect races in 100 runs and set one hour as timeout for it. If NodeRacer does not detect a known race in one hour, we regard that NodeRacer does not detect the race.

Column 7 of Table IV indicates whether the known race is detected by NodeRacer. As shown in Table IV, NodeRacer can only detect 6 out of 10 within the given timeout.

We investigate why NodeRacer fails to find four bugs in our experiment and find three reasons. First, although NodeRacer eliminates the schedule space with the help of happens-before relations, it can still miss races because of the large schedule space. For example, there are 277 events in project *socket.io*. Under the guidance of happens-before relations, the number of possible event schedules is 24,640. So, 100 runs only explores 0.4% of the schedule space. Second, NodeRacer only fuzzes the execution of events but does *not* perturb the execution of asynchronous tasks, which further reduces the possibility to expose races on external resources, e.g., the missing races in project *node-logger-file* and *xlsx-extract*. Further, NodeRacer can miss races if it runs out of time, e.g., the missing race in project *agentkeepalive*. This result demonstrates that NRace is more effective than NodeRacer to detect races in real-world Node.js applications.

TABLE V
DETECTION RESULT ON DATASET-2

ID	Project	#Total	#HRace	#BRace	#FP
1	nedb	2	0	1 (0)	1
2	node-http-proxy	0	0	0 (0)	0
3	baobab	0	0	0 (0)	0
4	simplecrawler	5	0	4 (2)	1
5	serve-static	0	0	0 (0)	0
6	nodejs-websocket	2	0	2 (1)	0
7	ncp	2	0	2 (2)	0
8	line-reader	2	0	2 (1)	0
9	json-file-store	1	1	0 (0)	0
10	fiware-pep-steelskin	13	5	6 (3)	2
	Total	27	6	17 (9)	4

C. Detect Unknown Races

We run NRace on dataset-2 to measure the NRace's ability to detect previously unknown races. Column *#Total* in Table V shows the number of detected races. As shown in Table V, NRace detects 27 races on dataset-2 in total. After manual inspection, we classify detected races into three categories: harmful races, benign races and false positives, the number of which is shown in column *#HRace*, *#BRace* and *#FP*, respectively. Similarly, the number of benign races automatically identified by NRace is shown in bracket in column *#BRace*.

NRace detects 6 harmful races in project *json-file-store* and *fiware-pep-steelskin*. For example, in *json-file-store*, event e_{sv} saves data with id $myId$ into the JSON file and event e_{rm} deletes data with the same id. The unordered event e_{sv} and e_{rm} make the content of the JSON file non-deterministic. We have submitted these detected races to developers. Five races in project *fiware-pep-steelskin* is acknowledged by developers.

Table V shows that NRace reports 4 false positives, which are also caused by ad-hoc synchronization, as discussed in dataset-1.

NRace detects 17 benign races. For example, in project *nodejs-websocket*, event e_{beg} asks the connection to begin transmitting data and event e_{cls} closes the connection. Consider event e_{cls} is executed before e_{beg} to set variable $state$ to $CLOSING$. When event e_{beg} is executed, it reads variable $state$ and finds variable $state$ does not equal to $OPEN$, it does not send data. Therefore, the race between event e_{beg} and e_{cls} has no harmful impact on the application. NRace automatically identifies 9 out of 17 benign races. More commutative patterns could be designed to enhance our approach.

D. Overhead

In Table VI, columns 2-5 report the performance of NRace, where column *TC*, *HBC*, *DP* and *T* denotes the time for trace collection, happens-before graph construction, offline race detection along with pruning, and the total time calculated by $TC + HBC + DP$, respectively.

As shown in Table VI, NRace detects races within a few seconds. The overhead in NRace is introduced by trace collection and happens-before graph construction, For example, in

TABLE VI
RUNTIME OVERHEAD ON DATASET-1

ID	NRace (s)				NodeRacer (s)		Speedup	
	TC	HBC	DP	T	HBC	T	HBC	T
1	1.0	0.2	0.1	1.3	2.0	3,600.0	10	2,769
2	1.0	7.0	1.0	9.0	104.5	1,665.5	15	185
3	0.5	0.1	0.1	0.7	0.4	1,279.8	4	1,828
4	0.5	0.1	0.1	0.7	0.1	351.2	1	502
5	1.3	0.2	0.1	1.6	1.0	743.6	5	465
6	6.6	1.3	0.5	8.4	7.8	87.5	6	10
7	0.5	0.4	0.1	1.0	35.5	945.3	89	945
8	0.6	0.1	0.1	0.8	0.1	282.6	1	353
9	5.5	0.7	0.7	6.9	5.7	1,203.6	8	174
10	1.2	14.3	0.3	15.8	215.5	465.5	15	29
Total	18.7	24.4	3.1	46.2	372.6	10,624.6	15	230

T: Trace collection; B: Happens-before graph construction; DP: Race detection and pruning; T: Total time.

project *xlsx-extract*, which consumes the most time on happen-before graph construction, there are 675 events, 290 of which are *Immediate* events. As discussed before, FIFO and NextTick rules need to be applied to *Immediate* events. It takes much time to repeatedly evaluate these two rules.

We compare the performance of NRace with NodeRacer on dataset-1. In Table VI, columns 6-7 present the overhead of NodeRacer, where column *HBC* and *T* denotes the overhead of the happens-before graph construction algorithm and the total runtime, respectively. Columns 8-9 show NRace's speedup compared with NodeRacer, where column *HBC* and *T* denotes the speedup in happens-before graph construction and the total runtime, respectively.

On the one hand, our proposed happens-before graph construction algorithm performs much faster than the algorithm of NodeRacer (from 1X to 89X). Since project *node-mkdirp* and project *del* have a small number of events, there is little difference between the overhead of our proposed algorithm and NodeRacer's algorithm in these two projects. Our proposed algorithm is efficient in happens-before graph construction for Node.js applications.

One the other hand, since NodeRacer repeatedly executes the application, it has large overhead. As shown Table VI, it spends nearly three hours to detect 6 races. As shown in column 9, NRace performs much faster than NodeRacer (from 10X to 2,769X). These results demonstrate that NRace is more efficient in detecting races in real-world Node.js applications.

E. Threats to Validity

The threat to validity is the representativeness of experimental projects. First, races selected in the evaluation come from real-world Node.js applications and have been studied by previous work NodeCB [10] and NodeRacer [21]. We believe these races represent real-world bugs. Second, since most of our selected Node.js applications have an amount of stars, we believe they are popular and representative.

V. RELATED WORK

In this section, we discuss related works close to ours.

Event race detection. CAFA [8] and DroidRacer [18] build happens-before relations among events, which account for only one event queue. EventTrack [29] maintains a subset of happens-before relations to optimize happens-before graph construction. SIERRA [19] reifies threads, events and user actions as actions and statically builds happens-before relations to improve precision. These existing approaches on Android applications cannot deal with multi-priority event queues.

WebRacer [7] formalizes the happens-before relations with web features. EventRacer [13] proposes race coverage to reduce the number of false positives and takes advantage of chain decomposition to decrease the overhead of reachability query. WAVE [14] records a sequence of operations and controls the target program to execute the observed operations to detect event races. ARROW [30] further statically detects event races and automatically repairs them. These approaches mainly focus on programming model features of browsers, such as DOM and AJAX. Therefore, they cannot be applied on Node.js applications.

Concurrency bug detection on Node.js applications. NodeCB [10] presents an empirical study on 57 concurrency bugs on real-world Node.js applications, and shows light on concurrency bugs in Node.js applications. [31] proposes a parallel programming abstraction GEMs. [27] proposes Async Graph to reason about event behaviors. NodeAV [22] can detect atomicity violations and does not consider other kinds of races. Node.fz [11] and NodeRacer [21] fuzzes the execution order of events to expose races. Due to randomness, they cannot deterministically find bugs. In contrast, our approach can systematically explore all the scheduling space of events and asynchronous tasks.

VI. CONCLUSION

Node.js applications are increasingly popular and are widely used by many developers and industrial giants. These applications are written in an asynchronous event-driven architecture, and suffer from races. In this paper, we propose NRace to detect races based on an observed execution trace. We build precise happens-before relations among events and asynchronous tasks in Node.js applications, which supports multi-priority event queues. We further develop a predictive race detection technique based on happens-before relations. We evaluate NRace on real-world Node.js applications and experimental results show it can detect known races as well as unknown races.

VII. ACKNOWLEDGE

We thank Yushan Zhang and Yu Gao for providing insightful comments about this work. This work was partially supported by National Key R&D Program of China (2017YFB1001804), National Natural Science Foundation of China (61732019, 62072444), Foundation of Science and Technology on Parallel and Distributed Processing Laboratory (61421102000402), Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036), and Youth Innovation Promotion Association at Chinese Academy of Sciences (2018142).

REFERENCES

[1] The npm repository. [Online]. Available: https://www.npmjs.com/

[2] 64 Node.js stats that prove its awesomeness in 2021. [Online]. Available: https://hostingtribunal.com/blog/node-js-stats/

[3] Node.js at PayPal. [Online]. Available: https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/

[4] Node.js at Uber. [Online]. Available: https://foundation.nodejs.org/wp-content/uploads/sites/50/2017/09/Nodejs-at-Uber.pdf

[5] Node.js at Yahoo. [Online]. Available: https://www.joyent.com/blog/node-js-on-the-road-boston-node-js-at-yahoo

[6] Libuv. [Online]. Available: https://github.com/libuv/libuv

[7] B. Petrov, M. T. Vechev, M. Sridharan, and J. Dolby, "Race detection for web applications," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012, pp. 251–262.

[8] C. Hsiao, C. Pereira, J. Yu, G. Pokam, S. Narayanasamy, P. M. Chen, Z. Kong, and J. Flinn, "Race detection for event-driven mobile applications," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014, pp. 326–336.

[9] P. Bielik, V. Raychev, and M. T. Vechev, "Scalable race detection for Android applications," in *Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015, pp. 332–348.

[10] 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 IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 520–531.

[11] J. C. Davis, A. Thekumparampil, and D. Lee, "Node.fz: Fuzzing the server-side event-driven architecture," in *Proceedings of European Conference on Computer Systems (EuroSys)*, 2017, pp. 145–160.

[12] Y. Zheng, T. Bao, and X. Zhang, "Statically locating web application bugs caused by asynchronous calls," in *Proceedings of the International Conference on World Wide Web (WWW)*, 2011, pp. 805–814.

[13] V. Raychev, M. Vechev, and M. Sridharan, "Effective race detection for event-driven programs," in *Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013, pp. 151–166.

[14] S. Hong, Y. Park, and M. Kim, "Detecting concurrency errors in client-side JavaScript web applications," in *Proceedings of International Conference on Software Testing, Validation and Verification (ICST)*, 2014, pp. 61–70.

[15] E. Mutlu, S. Tasiran, and B. Livshits, "Detecting JavaScript races that matter," in *Proceedings of Joint Meeting of European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2015, pp. 381–392.

[16] C. Q. Adamsen, A. Møller, and F. Tip, "Practical initialization race detection for JavaScript web applications," *Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, vol. 1, pp. 66:1–66:22, 2017.

[17] C. Q. Adamsen, A. Møller, S. Alimadadi, and F. Tip, "Practical AJAX race detection for JavaScript web applications," in *Proceedings of Joint Meeting of European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018, pp. 38–48.

[18] P. Maiya, A. Kanade, and R. Majumdar, "Race detection for Android applications," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014, pp. 316–325.

[19] Y. Hu and I. Neamtiu, "Static detection of event-based races in Android apps," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018, pp. 257–270.

[20] Y. Hu, I. Neamtiu, and A. Alavi, "Automatically verifying and reproducing event-based races in Android apps," in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, 2016, pp. 377–388.

[21] A. T. Endo and A. Møller, "NodeRacer: Event race detection for Node.js applications," in *Proceedings of International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 120–130.

[22] 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 International Conference on Software Engineering (ICSE)*, 2019, pp. 631–642.

[23] M. C. Loring, M. Marron, and D. Leijen, "Semantics of asynchronous JavaScript," in *Proceedings of ACM SIGPLAN International Symposium on on Dynamic Languages (DLS)*, 2017, pp. 51–62.

[24] Promise. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

[25] Node.js documentation about async hooks. [Online]. Available: https://nodejs.org/api/async_hooks.html

[26] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for JavaScript," in *Proceedings of Joint Meeting of European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2013, pp. 488–498.

[27] H. Sun, D. Bonetta, F. Schiavio, and W. Binder, "Reasoning about the Node.js event loop using async graphs," in *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, 2019, pp. 61–72.

[28] Cloc: Count lines of code. [Online]. Available: https://github.com/AlDanial/cloc

[29] P. Maiya and A. Kanade, "Efficient computation of happens-before relation for event-driven programs," in *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2017, pp. 102–112.

[30] W. Wang, Y. Zheng, P. Liu, L. Xu, X. Zhang, and P. Eugster, "ARROW: Automated repair of races on client-side web pages," in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, 2016, pp. 201–212.

[31] D. Bonetta, L. Salucci, S. Marr, and W. Binder, "Gems: Shared-memory parallel programming for Node.js," in *Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016, pp. 531–547.