Detecting Atomicity Violations for Event-Driven Node.js Applications

Xiaoning Chang, Wensheng Dou*, Yu Gao, Jie Wang, Jun Wei, Tao Huang
State Key Lab of Computer Sciences, *Institute of Software*, *Chinese Academy of Sciences*
University of Chinese Academy of Sciences
Beijing, China
{changxiaoning17, wsdou, gaoyu15, wangjie12, wj, tao}@otcaix.iscas.ac.cn

***Abstract*—Node.js has been widely-used as an event-driven server-side architecture. To improve performance, a task in a Node.js application is usually divided into a group of events, which are non-deterministically scheduled by Node.js. Developers may assume that the group of events (named *atomic event group*) should be atomically processed, without interruption. However, the atomicity of an atomic event group is not guaranteed by Node.js, and thus other events may interrupt the execution of the atomic event group, break down the atomicity and cause unexpected results. Existing approaches mainly focus on event race among two events, and cannot detect high-level atomicity violations among a group of events. In this paper, we propose *NodeAV*, which can predictively detect atomicity violations in Node.js applications based on an execution trace. Based on happens-before relations among events in an execution trace, we automatically identify a pair of events that should be atomically processed, and use predefined atomicity violation patterns to detect atomicity violations. We have evaluated NodeAV on real-world Node.js applications. The experimental results show that NodeAV can effectively detect atomicity violations in these Node.js applications.**

Keywords—Node.js, event-driven architecture, atomicity violation, happens-before

I. INTRODUCTION

Node.js [1] is a popular event-driven framework for developing server-side JavaScript applications. Since its birth in 2009, Node.js has caught much attention and becomes one of the leading open-source projects, like Linux, Git and MySQL [2]. Thanks to Node.js, JavaScript has become a widely-used server-side programming language. One evidence is that, the package ecosystem in Node.js, *npm* [3], has managed 700,000 building blocks in August 2018, which is the largest package registry so far. Node.js has also been widely used in industry, such as PayPal [4], LinkedIn [5], Yahoo [6] and Mozilla [7].

Node.js adopts an event-driven architecture, and provides effective asynchronous programming model. In traditional multithreaded programming model, a thread has to wait until an I/O operation is completed. Thus, much time is wasted on waiting for I/O operations and the performance may be degraded. In Node.js, a time-consuming I/O operation, such as file read and write, can be delegated as an asynchronous I/O operation, which runs in the dedicated underlying threads. Once the asynchronous I/O is completed, an I/O completion event is put into the event loop, and will be processed later by the looper thread (i.e., the main thread in Node.js). Therefore, the looper thread can continue to process other events without waiting for the I/O completion.

```
1. setImmediate (function cbImmediate(){
2.     fs.exists ('/tmp.txt',(function cbExist (exists){
3.         if (exists)
4.             fs.readFile ('/tmp.txt', …);
5.     });
6. });
7.  setTimeout ((function cbTimeout(){
8.      fs.unlink('/tmp.txt', …);
9.  },5);
```

Fig. 1. A Node.js example. The APIs *setImmediate* and *setTimeout* are used to trigger the execution of callback *cbImmediate* and *cbTimeout*.

To avoid blocking the looper thread and thus degrading the application performance, developers need to delegate heavy computations and I/O operations into asynchronous operations. Thus, a task will be separated into a group of individual events. Since these events collaborate to finish the special task, developers may assume that these events should be processed consecutively, and no relevant events can interrupt their execution. That's said, these events should be processed all together, or neither of them should be processed. We name such a group of events as an *atomic event group*. For example, in Fig. 1, event *cbImmediate* first checks whether file */tmp.txt* exists by calling asynchronous I/O operation *fs.exists* (Line 2). When this asynchronous I/O is done, event *cbExist* begins to execute and reads file */tmp.txt* by calling *fs.readFile* (Line 4). Developers assume operation *fs.readFile* is protected by operation *fs.exists*. Therefore, event *cbImmediate* and *cbExist* should be processed all together, without interruption.

However, the event-driven architecture in Node.js does not have an effective mechanism that helps developers guarantee the atomicity of an atomic event group, and avoids interruption during processing the atomic event group. Thus, other relevant events may break the atomicity of an atomic event group. For example, as illustrated by the dotted line in Fig. 1, event *cbTimeout* can be executed between *cbImmediate* and *cbExist*. The asynchronous I/O operation *fs.readFile* (Line 4) will try to read the file that has been deleted by *cbTimeout*, and returns an error. Thus, an atomicity violation occurs. Recent studies on Node.js [8][9] have shown that atomicity violations are common, and at least 65% of concurrency bugs in Node.js are atomicity violations. Further, atomicity violations can cause severe consequences, e.g., exceptions and no response.

Existing approaches on atomicity violation detection [10][11][12][13] mainly focus on multithreaded programs. However, Node.js atomically processes each event in the event looper thread, and there does not exist atomicity violations

* Corresponding author

among threads. For event-driven architectures, e.g., Android and client-side JavaScript applications, researchers have proposed many interesting approaches [14][15][16][17][18][19][20][21] to detect event races, in which two events access to the same resource (at least one is write), and can be processed in any order. First, Node.js differs from these systems as they originate from different programming paradigms and execution environments. For example, Android mostly concerns the Android GUI model and asynchronous tasks executed in other threads [14][15], and client-side JavaScript applications mostly concern about the features like DOM and AJAX [16][17], while Node.js does not have such features. Second, atomicity violations concern a group of events, and the atomicity intentions of developers. This is different from event races. Thus, existing approaches cannot apply to atomicity violation detection in Node.js applications.

In this paper, we propose *NodeAV*, a dynamic atomicity violation detector for Node.js applications. After collecting the execution trace of a Node.js application, NodeAV predictively infers possible atomicity violations. NodeAV faces two major challenges. (1) How can we infer the atomicity intentions of developers? Node.js does not have any mechanism to express the atomicity intentions of developers, and thus atomicity intentions are usually not documented. We observe that, if two events form an event processing chain and access the same resource, they usually logically belong to the same task, and should be processed together, without interruption. For example, in Fig. 1, event *cbImmediate* and *cbExist* form an event processing chain and access the same file */tmp.txt*. Thus, we regard them as an *atomic event pair* intended by developers. (2) How can we identify interleaving events that can cause atomicity violations? Not all interleaving events that occur among an atomic event pair can cause atomicity violations. For example, in Fig. 1, suppose that event *cbTimeout* reads file */tmp.txt* instead of deleting the file, no atomicity violation occurs because the interleaving file reading does not affect the result of the atomic event pair. We adapt unserializable schedules in multithreaded programs [10] on Node.js applications and summarize the atomicity violation patterns to detect atomicity violations in Node.js applications.

We summarize the main contributions as follows:

- We propose a novel approach to infer atomicity intentions among events in Node.js applications, based on happens-before model for Node.js.
- We design an automated approach to detect atomicity violations in Node.js applications, by identifying atomicity violation patterns in them.
- We implement our approach as a tool NodeAV and evaluate it on real-world Node.js applications. The experimental results show that NodeAV can detect atomicity violations in Node.js applications effectively.

The remainder of this paper is organized as follows. Section II presents related background and our motivation. Section III introduces our approach. Section IV describes implementation details. Section V evaluates our approach experimentally. Section VI discusses threats to validity and limitations of our approach. Section VII and VIII discuss related work and conclude this paper, respectively.

II. Background and Motivation

In this section, we illustrate the event-driven programming model in Node.js and atomicity violations occurring in a real-world Node.js application. Then, we discuss the challenges in detecting atomicity violations in Node.js applications.

A. Event-Driven Programming Model in Node.js

The event-driven programming model in Node.js mainly consists of two parts: a single looper thread and a worker pool. The looper thread fetches *events* from its *event queues*, and executes their associated *callbacks*. For expensive operations, e.g., file read, Node.js delegates them to the worker pool, and executes them asynchronously.

Event: An event in Node.js can be generated by network traffic (e.g., user request), timers, the completion of asynchronous operations, and platform APIs (e.g., *process.nextTick* and *setImmediate*). An event is processed by invoking its *callback*, which is registered associated with the event. According to the official Node.js documents [22], events can be categorized into five types according to their sources: *Timeout*, *Immediate*, *nextTick*, *promise*, and *IO*, whose events are generated by *setTimeout*(), *setImmediate*(), *process.nextTick*(), *promise*, and *asynchronous I/O*, respectively. Events with different types are put into their corresponding event queues in Node.js.

Event queue: Once an event is generated, it is put into an event queue. Node.js consists of seven event queues that hold different types of events: *timers*, *I/O*, *pending*, *idle*, *prepare*, *check*, and *close* [22]. For each event queue, its events are processed in the order that they are enqueued. Node.js provides mechanisms to prioritize events in different event queues. For example, an event scheduled by *process.nextTick* will be processed immediately after the current event is processed.

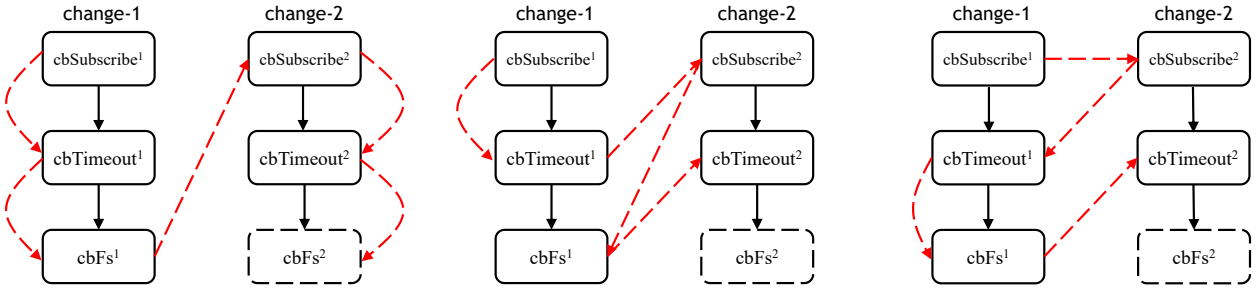
Looper thread: The looper thread continuously checks the event queues, selects one event to process at a time. In Node.js, there is only one looper thread. Thus, each event is guaranteed to be processed atomically, without interruption. In Node.js, the looper thread processes the above seven event queues in a round-robin manner: when a queue has been exhausted or the amount of the executed callbacks for a queue reaches a given threshold, the looper thread will move on to the next queue.

Asynchronous operation: Expensive operations are delegated to the worker pool, and executed asynchronously. When the asynchronous operations are done, a 'operation done' event will be put into the event queues, and consumed later by the looper thread. By offloading expensive operations to the worker pool, the looper thread will not be blocked by expensive operations.

Callback chain: To avoid blocking the looper thread, a heavy task is usually divided into multiple steps $\{s_1, s_2, \ldots, s_n\}$, which are connected by intermediate events and their associated callbacks $\{cb_1, cb_2, \ldots, cb_n\}$. These callbacks form a callback chain.

B. Motivating Example

Fig. 2 shows the simplified code snippet extracted from a real-world Node.js application, change propagation [23], which contains an atomicity violation reported in [24].



(a) Correct trace τ_1 (b) Buggy trace τ_2 (c) Buggy trace τ_3

Fig. 3. Three execution traces for the code snippet in Fig. 2. The solid boxes represent events and the dotted boxes represent virtual events that will not happen in the execution. The solid arrows denote the happens-before relation among events, and the dotted arrows denote the execution order. In τ_2, *cbFs*[1] that sets variable *x.msg* to *undefined* causes the succeeding *cbTimeout*[2] to fail. In τ_3, *cbSubscribe*[2] that overwrites variable *x.msg* written by *cbSubscribe*[1] causes the message for *change*-1 lost.

```
1.  var x={msg: undefined};
2.  subscribe (someTopic).then (function cbSubscribe (message) {
3.      x.msg=message;
4.      setTimeout (function cbTimeout () {
5.          if (x.msg)
6.              fs.writeFile ('log.txt', x.msg, function cbFs () {
7.                  x.msg=undefined;
8.              });
9.      }, 100);
10. });
```

Fig. 2. Code snippet of an atomicity violation.

This code snippet is used to process topic changes. In this example, users can subscribe a topic by calling *subscribe(someTopic)* and a callback *cbSubscribe* is registered to process the topic changes by calling *then* method (Line 2). When the topic changes, callback *cbSubscribe* is triggered with the parameter *message* that stores the topic change information. It copies the value of *message* to *x.msg* (Line 3) and then registers a callback *cbTimeout* by *setTimeout* API (Line 4). When the timer expiration event happens, callback *cbTimeout* is triggered. If the value of *x.msg* is valid (Line 5), *cbTimeout* offloads an asynchronous I/O operation to the worker pool to write the variable *x.msg* to file *log.txt* by calling the method *writeFile* (Line 6). When the file write is completed, callback *cbFs* registered by *fs.writeFile* (Line 6) is triggered to clear the content of *x.msg* (Line 7).

In this example, the processing for a topic change is divided into three callbacks, i.e., *cbSubscribe*, *cbTimeout* and *cbFs*, which forms a callback chain. For each topic change, the execution order of *cbSubscribe*, *cbTimeout* and *cbFs* is deterministic, namely *cbSubscribe* → *cbTimeout* → *cbFs*. The callback chain is used to process the specific task, and the atomicity of the callback chain should be guaranteed. For example, *cbSubscribe* and *cbTimeout* access the shared variable *x.msg*, hence they should be executed consecutively, without interruption. Although the execution order among the events (i.e., callbacks) in a callback chain is deterministic, the atomicity of the callback chain is not guaranteed by Node.js. Let's assume that two topic changes *change*-1 and *change*-2 simultaneously arrive. The callback chain will be executed twice, once per request. For simplicity, we denote an execution of callback *cb* for request *change-x* as cb^x ($x \in \{1, 2\}$). Fig. 3a shows a correct execution trace, in which two changes are processed one by one.

However, the two executions of *change*-1 and *change*-2 can interleave and thus introduce atomicity violations. In the buggy interleaving execution τ_2 in Fig. 3b, *cbFs*[1] interleaves between *cbSubscribe*[2] and *cbTimeout*[2], and sets variable *x.msg* to *undefined*. Thus, the succeeding *cbTimeout*[2] switches to a different branch in Line 5 and does not save the message in *change*-2. Fig. 3c shows another buggy interleaving execution τ_3, in which, *cbSubscribe*[2] interleaves between *cbSubscribe*[1] and *cbTimeout*[1], *and* overwrites the message in *change*-1 (Line 3). Thus, the message in *change*-1 is lost.

C. Approach Overview

In this paper, we propose NodeAV to dynamically detect atomicity violations in Node.js applications. We need to address four technical challenges. (1) How can we model the contention on external resources, e.g., the file in Fig. 1? Existing studies [8][9] have shown that about a half of concurrency bugs in Node.js applications contend against external resources. (2) Node.js has various mechanisms for event-driven programming, including several system-specific scheduling APIs, e.g., *process.nextTick* and *promise* [22][25]. How can we build a precise happens-before relation among events based on an collected execution trace? (3) The atomicity intentions of developers are usually not documented explicitly. What events can form an atomic event group? (4) In what situation, some events can cause atomicity violations of atomic event groups?

For the first challenge, we summarize the file system APIs in Node.js into basic access types, and regard each API as one or multiple basic access types, e.g., read and write. For the second challenge, we design happens-before model to capture partial order among events in Node.js applications. For the third challenge, automatically checking which events can form an atomic event group is a hard problem, as it depends on the high-level semantics of events involved in an application. Instead, we only check whether two events (atomic event pair) should be processed together without interruption (i.e., atomically). We require that (a) the two events access the same resource (e.g., shared variables or files) and (b) the execution order of two events can be determined by happens-before relations (e.g., callback chain in Fig. 3). For the fourth challenge, we determine whether an atomic event pair and a related interleaving event can be serializable. We borrow the idea from the serializability in multithreaded programs [10], and apply atomicity violation patterns on event-driven Node.js applications.

III. Approach

Given the source code of a Node.js application and its test suite, NodeAV detects atomicity violations in three steps. First, it instruments the source code, and then executes the test suite on the instrumented version to collect execution trace, including events, read/write to variables, and so on (Section A). Second, we design happens-before model for Node.js and build a happens-before graph for events in the execution trace, which reflects the partial order among events (Section B). Third, we infer atomic event pairs based on happens-before graphs, and detect atomicity violations based on predesigned atomicity violation patterns (Section C).

A. Execution Trace

A Node.js application's execution consists of a number of events, i.e., the execution of events' associated callbacks. Thus, an execution trace of a Node.js application is a sequence of operations that are performed by events in the execution.

The operations in an execution trace are listed in Fig. 4. Note that, we only consider these operations in Fig. 4, since other parts in Node.js applications, e.g., conditionals, loops and expressions, are irrelevant to atomicity violation detection and omitted for brevity.

- *start*(*e*) and *end*(*e*): The begin and end of an event *e*, respectively.
- *register*(*e*, *listener*): Event *e* registers a *listener*, which contains an expected event *listener.event*, and an associated *listener.callback*. When the expected event *listener.event* is triggered, *listener.callback* will be executed. When registering a listener, the type of its expected event *listener.event.eType* is determined. For example, *fs.readFile*('*log.txt*', *cb*) registers a listerner that listens on the file read done event with the type of *IO*. As discussed earlier in Section II.A, Node.js provides several APIs to register listeners with different expected event types, e.g., *process.nextTick* and *setImmediate*. We map these APIs into *register* operations with their expected event types.
- *trigger*(*e*, *u*): Event *e* triggers an event *u* so that event *u* is put into the corresponding event queue, and its associated callback will be executed. In Node.js applications, developers can use system-specific APIs to generate specific events, e.g., *process.nextTick*, *setImmediate*, and function *resolve* in *promise* [25]. When an event is triggered, it will be put into the corresponding event queue according to its event type.
- *trigger*(*libuv*, *u*): Node.js underlying platform (i.e., *libuv* [26]) can generate timeout events and I/O events, etc. For example, a file read done event is triggered by *libuv*, and its triggering timing is unknown. When an event is triggered by *libuv*, it will be put into the related event queue according to its event type.
- *access*(*e*, *resource*, *type*): Event *e* accesses to a shared *resource* with a specific accessing *type*.

Shared resources are accessed by multiple events. In Node.js, shared resources include variables (e.g., variable *x.msg* in Fig. 2) and external resources (e.g., file '*tmp.txt*' in Fig. 1). For each shared resource, its accessing type can be generally abstracted into two types: *read* and *write*. We describe accessing types for three kinds of shared resources in the following.

$Trace \rightarrow Operation^*$
$Operation \rightarrow start(e) \mid end(e) \mid$
$register(e, listener) \mid trigger(e, u) \mid trigger(libuv, u)$
$access(e, resource, type)$
$e, u \in Event$
$resource \in$ Shared variables $\cup$ files
$type \in \{read, write\}$

Fig. 4. Operations in a Node.js execution trace.

- Variable. Reading variables and getting field of objects are considered as an operation of type *read*. Writing variables and putting fields of objects are considered as an operation of type *write*. Specially, functions are treated as a special kind of objects. We regard a declaration of a function *f* as an operation of type *write* on variable *f* and a call of the function *f* as an operation of type *read* on variable *f*.
- Native object. JavaScript predefines a large number of native objects [27], such as *Array* and *String*. The properties of native objects are accessed by native methods. We study the semantics of native methods and map them into one or multiple operations of type *read* and *write* on native objects.
- File. We map each file system API into one or multiple operations of type *read* and *write* on files. For instance, *fs.readFile(foo, ...)* is mapped into an operation of type *read* on file *foo* and *fs.copyFile(src, dest, ...)* is mapped into an operation of type *read* on file *src* and an operation of type *write* on file *dest*.

B. Happens-Before Relaion

Given an execution trace τ for a Node.js application, happens-before relation $\prec$ is a partial order among events in the execution trace. We denote event e_1 happens before event e_2 as $e_1 \prec e_2$.

For brevity, we also denote an operation *a* happens before *b* as $a \prec b$, an operation *a* happens before all operations in event *e* as $a \prec e$, and all operations in event *e* happen before an operation *a* as $e \prec a$. Here, we overload the happens-before operator for operations and events. We present our happens-before rules in Node.js applications as follows.

Rule 1 (Transitivity): The happens-before relation $\prec$ is transitive, i.e., if $e_1 \prec e_2$ and $e_2 \prec e_3$, then $e_1 \prec e_3$.

Rule 2 (Program order): If operations *a* and *b* are performed by the same event *e* and *a* occurs before *b* in event *e*, then $a \prec b$.

Rule 3 (Event atomicity): In Node.js, an event is processed without interruption by the looper thread. In other words, an event is either processed atomically or not processed. Namely, for any operation *a* in event e_1 and any operation *b* in event e_2, if *a* happens before *b*, then any operation in event e_1 happens before any operation in event e_2. Formally:

$$\text{if } start(e_1) \prec end(e_2), \text{then } end(e_1) \prec \text{start}(e_2)$$

Rule 4 (Event register): If event e_1 performs an operation *register*(e_1, *listener*) and event e_2 triggers the *listener* and executes its associated callback, then $e_1 \prec e_2$. Formally:

$$\forall\, register(e_1, listener), \text{if } e_2 = listener.event, \text{ then } e_1 \prec e_2$$

Rule 5 (Event trigger): If event e_1 performs an operation *trigger*(e_1, e_2) to put event e_2 into its corresponding event queue, then $e_1 \prec e_2$. Formally:

$$\forall\, trigger(e_1, e_2), e_1 \prec e_2$$

Rule 6 (Event trigger by *libuv*): As discussed in Section III.A, *libuv* (i.e., Node.js underlying platform) can generate timeout events and I/O events, etc. These trigger operations issued by *libuv* are non-deteministic. However, we can restrict their happens-before relation with other operations. If event e_1 performs an operation *register*(e_1, *listener*) and event e_2 is triggered by *libuv*, and associates with the *listener*, then operation *register*(e_1, *listener*) happens before *trigger*(*libuv*, e_2) and *trigger*(*libuv*, e_2) happens before e_2. Formally:

$$\forall\, register(e_1, listener) \text{ and } trigger(libuv, e_2), \text{ if } e_2 = listener.event, \text{ then } register(e_1, listener) \prec trigger(libuv, e_2) \prec e_2$$

As discussed earlier in Section II.A, events with different types are put into different event queues, and the looper thread processes event queues in a round-robin manner. According to the Node.js official documents [22] and implementation, events in different event queues have different priorities to be processed by the looper thread. We assign priorities to events according to event types. Basically, there are four levels of priorities.

- 0 for *nextTick* and *promise* events, triggered by *process.nextTick* and *promise*.
- 1 for *Immediate* events, triggered by *setImmediate*.
- 2 for *Timeout* events, triggered by *setTimeout*.
- 3 for *IO* events, triggered by asynchronous I/O.

Rule 7 (Events with the same priority): Events in the same event queue have the same priority, and are processed in their enqueue order. If an operation *trigger*(e_1, u_1) happens before another operation *trigger*(e_2, u_2) and the priority of event u_1 is the same as the priority of event u_2, then $u_1 \prec u_2$. We use *priority*(e) denotes the priority of an event e. Formally:

$$\text{if } trigger(e_1, u_1) \prec trigger(e_2, u_2) \text{ and } priority(u_1) = priority(u_2), \text{then } u_1 \prec u_2$$

Rule 8 (Events with different priorities): According to the Node.js official documents [22] and implementation, Node.js basically adopts a round-robin manner to process events with different priority in event queues. Therefore, Node.js does not guarantee that an event with higher priority will be scheduled earlier, e.g., a *Timeout* event may be scheduled before an *Immediate* event, even the previous one has a higher priority. The Node strategy in [28] precisely models the Node.js scheduling: If there exist events of priority 0, then Node.js executes all these events with priority 0 recursively; Otherwise, for events with other priorities (i.e., 1, 2 and 3), Node.js picks one event to execute, regardless of its priority.

```
1. var x;
2. setImmediate (function cb1 () {
3.     x=1;
4. });
5. setImmediate (function cb2 () {
6.     if (x==1) …
7. });
```

Fig. 5. Two events with predesigned execution order.

Suppose that event e_1 performs an operation *trigger*(e_1, u_1) before and e_2 performs an operation *trigger*(e_2, u_2), and event u_1 and u_2 have different priorities. There are three cases.

Case 1: event u_1 is of priority 0. Formally:

$$\text{if } trigger(e_1, u_1) \prec trigger(e_2, u_2), priority(u_1) = 0 \text{ and } priority(u_2) \neq 0, \text{then } u_1 \prec u_2$$

Case 2: event u_2 is of priority 0. In this case, if the *trigger*(e_2, u_2) occurs before the execution of u_1, then $u_2 \prec u_1$. Formally:

$$\text{if } trigger(e_1, u_1) \prec trigger(e_2, u_2), priority(u_1) \neq 0, priority(u_2) = 0 \text{ and } trigger(e_2, u_2) \prec begin(u_1), \text{ then } u_2 \prec u_1$$

Case 3: Neither event u_1 or u_2 are of priority 0. In this case, the happens-before relation of u_1 and u_2 cannot be determined.

C. Atomicity Violation Detection

In Node.js, developers usually divide a task into a group of events, which are non-deterministically scheduled by Node.js. Since this group of events should collaborate to finish the special task, developers usually assume that these events are processed consecutively without interruption. However, the atomicity of this event group is not guaranteed by Node.js, and other events may interrupt the execution of atomic event group, break down the atomicity.

The key to atomicity violation detection in Node.js is to automatically infer which events can form atomic event groups. Developers' atomicity intentions of events involve high-level the semantics of the application and are not documented explicitly. It is hard to infer these high-level atomicity intentions. However, we have two observations which help us determine whether some events should be executed without interruption.

- Callback chain: As discussed earlier in Section II.A, a callback chain clearly describes how a task is performed by a sequence of events. This usually reflects that the events in the callback chain should not be interrupted. For example, in Fig. 3a, event *cbSubscribe*[1] and *cbTimeout*[1] are intended to be executed together to process task *change*-1.
- Predesigned execution order of events: The execution orders of two or more events can be clearly designed in the application. For example, in Fig. 5, the events triggered by two *setImmediate* operations are ordered by Rule 7 in Section III.B. Developers assume that two consecutive events *cb1* and *cb2* should not be broken down.

According to the above two observations, we find that developers usually schedule an ordered sequence of events to deal with a task, and assume events in the ordered sequence belong to an atomic event group. The execution order of events can be precisely expressed by the happens-before relation in

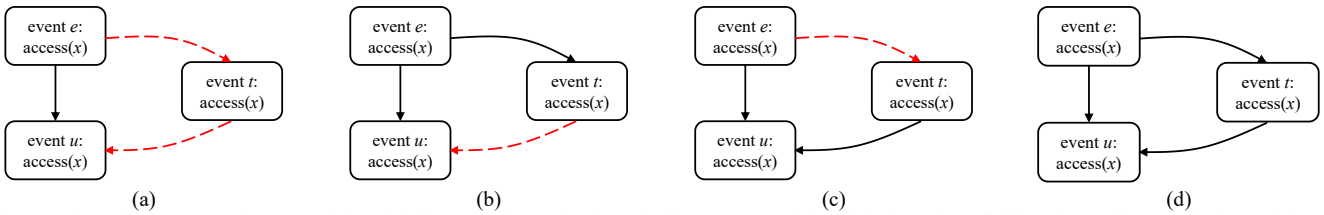


Fig. 6. Subgraph *a-c* present three atomicity violation scenarios and subgraph *d* is not an atomicity violation. The solid lines denote happens-before relation and the dotted lines denote execution order.

Section III.B. However, conservatively treating all events involved into happens-before relation will result into that all events in an execution trace are treated as an atomic event group. It is because the start event (similar to the main function) in an application happens before all other events in an execution. This motivates us to design a general heuristic to infer atomic event pairs, rather than atomic event groups.

Atomic event pair: Given event *e* and *u*, if *e* happens before *u*, and *e* and *u* have some operations op_1 and op_2 that access the same resource, then *e* and *u* forms an atomic event pair, denoted as *atomic*(*e*, *u*). We use *resource*(*op*) to denote the resource that operation *op* accesses. Formally, event *e* and *u* in an atomic event pair satisfy the following two conditions.

- $e \prec u$
- $\exists op_1 \in e, op_2 \in u, resource(op_1) = resource(op_2)$

The first condition requires that events in an atomic event group are usually ordered. The second condition indicates that if two events do not share any resource, they can be considered as independent events, and we do not need to force the atomicity among them. Note that, two events that are not ordered by happens-before relation are not treated as atomic event pair.

We further define atomicity violation as follows.

Atomicity violation: Given an atomic event pair *atomic*(*e*, *u*) and another event *t*, if event *t* can interrupt into this pair, and event *e*, *u* and *t* have some operations that access the same resource, then *atomic*(*e*, *u*) and event *t* form an atomicity violation. Formally, atomic event pair *atomic*(*e*, *u*) and event *t* should satisfy the following three conditions. Here, $e \nprec t$ denotes event *e* does not happen before event *t*.

- $e \prec u$
- $e \nprec t \wedge t \nprec e \vee u \nprec t \wedge t \nprec u$
- $\exists op_1 \in e, op_2 \in u, op_3 \in t, resource(op_1) = resource(op_2) = resource(op_3)$

The first condition requires that event *e* and *u* can be an atomic event pair. The second condition indicates that event *e* and *t* do not have happens-before relation, or event *u* and *t* does not have happens-before relation. So, event *t* can happen between event *e* and *u*. The application may run correctly when event *t* happens before or after the atomic event pair. The third condition indicates that three events share the same resource, otherwise, they can be considered as independent events.

Fig. 6a-c shows three scenarios in which atomicity violations can happen. In Fig. 6d, $e \prec t$ and $t \prec u$, so that event *e*, *t* and *u* must be executed in the order of $e \longrightarrow t \longrightarrow u$. Note that, even though this execution order is the same as those in atomicity violations of Fig. 6a-c, we do not consider event *e*, *t*, and *u* in Fig. 6d form an atomicity violation. It is because $e \longrightarrow t \longrightarrow u$ is the only execution order for these three events, and should be excepted execution order by developers.

Although three events *e*, *t* and *u* in Fig. 6a-c can form an atomicity violation, they do not necessarily incur different execution results. For example, in Fig. 7a, event *e*, *u* and *t* all access the shared resource *x* with the type *read*, so the execution result is not affected by this atomicity violation. In contrast, in Fig. 1, atomic event pair (*cbImmediate*, *cbExist*) is interrupted by event *cbTimeout*, which performs an operation of type *write* on the shared file (Line 8). This atomicity violation affects the execution result of the operation of type *read* in event *cbExist* (Line 8). If an atomicity violation does not affect the execution result of an application, developers do not need to fix it. Thus, we only focus on atomicity violations which can incur different execution results.

Inspired by existing studies on atomicity violations in multithreaded programs [10], we borrow the idea of serializability from multithreaded programs, and design atomicity violation patterns based on shared resource accessing types. Based on the atomicity violation patterns in multithreaded program [10], for each atomicity violation scenario in Fig. 6a-c, we design four resource accessing patterns that can cause different execution results. Since all the three scenarios in Fig. 6a-c have the same resource accessing patterns, we only use the atomicity violation scenario in Fig. 6a as an example. Its four resource accessing patterns are shown as Fig. 7b-e. These four resource accessing patterns are briefly described as follows.

- Read-Write-Read in Fig. 7b: The *read* in event *u* gets a different value of resource *x* with that in event *e*.
- Read-Write-Read in Fig. 7c: The *read* in event *u* cannot get the expected value of resource *x* written by event *e*.
- Write-Read-Write in Fig. 7d: The read in *t* gets the dirty data of resource *x*, which is written by event *e*.
- Read-Write-Write in Fig. 7e: The *write* of event *u* depends on the value of resource *x* read by event *e*, which is overwritten by event *t*.

NodeAV monitors the executions of a Node.js application, and predicts atomicity violations that can happen in the future under the same input, and outputs the three events involved in each atomicity violation. We briefly describe the NodeAV detection algorithm as follows.

Given an execution trace of a Node.js application, we build a happens-before graph among events based on the happens-

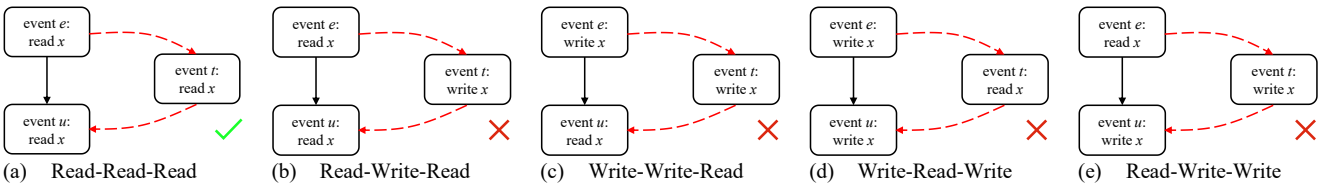


Fig. 7. The atomicity violation pattern in subgraph *a* does not affect the execution result, whereas the four atomicity violation patterns in subgraph *b*-*e* can affect the execution result. Event *e*, *u* and *t* share the resource *x* and their accessing types are denoted in each box. Event *e* happens before *u*, denoted by the solid line. They form an atomic event pair *atomic*(*e*, *u*). The dotted lines denote possible execution order.

before rules in Section III.B, in which, nodes denote events and edges denote happens-before relation. We construct the atomicity violation triple <e_1, e_2, e_3> in the following steps. (1) For any two event e_1 and e_2, if $e_1 \prec e_2$, we further check whether e_1 and e_2 access the same resource. If yes, they form an atomic event pair *atomic*(e_1, e_2). (2) We try to find an event e_3 that do not have edges with e_1 or e_2 in the happens-before graph. That's said, event e_3 do not have happens-before relation with e_1 or e_2. For each found e_3, we form a candidate atomicity violation triple <e_1, e_2, e_3>. (3) For each candidate triple, we validate whether they access the same resource in the patterns shown in Fig. 7b-e. If one pattern matches, the candidate triple is considered as an atomicity violation.

IV. Implementation

NodeAV first generates an execution trace by instrumenting the original Node.js application, and then performs atomicity violation detection offline on the execution trace. Here we mainly introduce the implementation details of trace generation, which is not described in Section III.

Node.js utilizes the *async_hooks* module [29], which is introduced in Node.js 8.6, to collect event scheduling operations, namely *start*(*e*), *end*(*e*), *register*(*e*, *listener*) and *trigger*(*e*, *u*). *async_hooks* provides four APIs to track the lifetime of each event: *init*, *before*, *after* and *promiseResolve*, which are triggered when a callback is registered, before and after an event and when *resolve* method is invoked. We use these APIs to record all above operations. In addition, *async_hooks* provides an API *executionAsyncId*, which returns the event that performs a given operation.

NodeAV utilizes Jalangi [30] to collect resource accessing operations. Jalangi interprets each instruction, such as *read* and *putField* instructions, and provides detailed information for each instruction, e.g., the name of a variable, and the value read from the variable. We use Jalangi to query the instructions that we are interested in, including *read*, *write*, *putField*, *getField*, *declare*, *invokeFun*, etc. For an operation *access*(*e*, *resource*, *type*), we obtain its information *e* by calling the method *async_hooks.executionAsyncId* and record the type of operation *type*. As discussed in Section III.C, we need to determine whether the resources accessed by two operations are the same. We use logic addresses to uniquely identify resources. We obtain logic addresses for different types of resources as follows.

- Variable. For a variable, we use a tuple <*scope*, *name*> as its logic address, where *scope* and *name* refer to the static scope and the name of the variable, respectively. For an object *obj*, when it is created, we allocate a unique ID *objId* to it. When its property *obj.prop* is accessed, we use a tuple <*objId*, *name*> to denote its logic address, where *name* refers to the name of the property *prop*.
- File. We use the absolute path of a file as its logical address. Jalangi provides an API *invokeFunPre*, which is called before a method invocation. We use *invokeFunPre* to check whether the method is a file system API. If yes, we use our file system API model to transform the API invocation into corresponding operations. In our file system API model, we specify how we transform each API into basic operations. For example, *fs.copyFile*(*src*, *dest*, …) is parsed into an operation with *read* type on *src* and an operation with *write* type on *dest*.

V. Evaluation

We evaluate NodeAV by answering the following two research questions:

RQ1: Can NodeAV effectively detect atomicity violations in Node.js applications?

RQ2: How is the performance of NodeAV? Can NodeAV scale to analyze real-world Node.js applications?

A. Experimental Subjects and Setup

We evaluate NodeAV on two datasets of real-world Node.js atomicity violations and applications.

Dataset-1: Known atomicity violations. To verify whether NodeAV can effectively detect known atomicity violations in Node.js applications, we build dataset-1 based on Node.fz [9] and NodeCB [8], which contain atomicity violations in real-world Node.js applications. We select an atomicity violation in Node.fz and NodeCB if it satisfies the following conditions: (1) The application that contains the atomicity violation is able to run on Node.js 8.6 or above, which supports the *async_hooks* module used by NodeAV. (2) The atomicity violation can be reproduced or has clear description in the bug report about how it occurs. If we cannot reproduce the original atomicity violation for some reasons, e.g., unavailable buggy version in *wikimedia/change-propagation*, then we design a standalone application to replicate the buggy code described in the bug report. The last three atomicity violations in are reproduced in Table I this way. (3) The application that contains the atomicity violation should have test cases so that we can use them to collect runtime traces. In case that the application does not contain related test cases, we develop a simple test case to trigger the actions described in the bug report. Finally, we obtain 9 atomicity violations from 8 applications, as shown in Table I. The column *Project* refers to the application, the column

TABLE I DATASET-1: KNOWN ATOMICITY VIOLATIONS. THE PROJECTS HAVE ACCESSIBLE LINKS.

ID	Project	Description	Issue ID
1	michaelwittig/node-logger-file	File endpoint for cinovo-logger	1
2	substack/node-mkdir-p	Similar function to mkdir -p	2
3	hapijs/nes	Native WebSocket support to hapi-based application servers	18
4	telefonicaid/fiware-pep-steelskin	Implementation of the FIWARE PEP GE	269
5	telefonicaid/fiware-pep-steelskin	Implementation of the FIWARE PEP GE	279
6	node-modules/agentkeepalive	Support keepalive http agent	23
7	wikimedia/change-propagation	A REST-based queuing module for Apache Kafka	84
8	brave/browser-laptop	Desktop browser for macOS, Windows, and Linux	3,273
9	gadicc/meteor-hmr	Hot module replacement for meteor	71

TABLE II DATASET-2: APPLICATIONS UNDER DETECTION. THE PROJECTS HAVE ACCESSIBLE LINKS.

ID	Project	Description	LoC	Star
1	michaelwittig/node-logger-file	File endpoint for cinovo-logger	367	2
2	substack/node-mkdir-p	Similar function to mkdir -p	76	1,218
3	hapijs/nes	Native WebSocket support to hapi-based application servers	1,616	146
4	telefonicaid/fiware-pep-steelskin	Implementation of the FIWARE PEP GE	1,655	9
5	node-modules/agentkeepalive	Support keepalive http agent	312	251
6	nickewing/line-reader	Asynchronous line-by-line file reader	256	384
7	sitegui/nodejs-websocket	A module for web socket server and client	726	514
8	JoshuaWise/better-sqlite3	The fastest and simplest library for SQLite3	285	1,050
9	derbyjs/racer	Realtime model synchronization engine	4,312	1,098
10	simplecrawler/simplecrawler	Flexible event driven crawler	285	1,847

TABLE III EXPERIMENTAL RESULT ON DATASET-1.

ID	Project	Detected	#New violation
1	michaelwittig/node-logger-file	Y	3
2	substack/node-mkdir-p	Y	0
3	hapijs/nes	Y	0
4	telefonicaid/fiware-pep-steelskin	Y	4
5	telefonicaid/fiware-pep-steelskin	N	0
6	node-modules/agentkeepalive	N	0
7	wikimedia/change-propagation	Y	1
8	brave/browser-laptop	Y	0
9	gadicc/meteor-hmr	Y	0

Description gives a brief description of each application and the column *Issue ID* shows the issue ID in GitHub.

Dataset-2: Real-world Node.js applications for detecting new atomicity violations. To evaluate whether NodeAV can find new atomicity violations in real-world Node.js applications, we collect 10 Node.js applications from two aspects. First, we update the applications in dataset-1 to their newest versions. We remove an application, if (1) it is incompatible with NodeAV, or (2) it does not have available test suites for execution trace collection, e.g., *brave/browser-laptop*, or (3) it cannot be built, e.g., *wikimedia/change-propagation* and *gadicc/meteor*. Thus, we obtain 5 Node.js applications from dataset-1, as shown in the first 5 rows in Table II. Second, we collect another 5 popular Node.js applications from GitHub, which exhibit certain concurrent behavior, e.g., having concurrency bug issues. They are shown in the last 5 rows in Table II. In Table II, column *Star* shows the number of stargazers in GitHub. We can see that most of our selected applications are popular.

We perform our experiments in the following steps. First, we use NodeAV to instrument the source code of an application. Second, we run the test cases included in the bug report or test cases developed by us and collect execution traces. Third, we use NodeAV to detect atomicity violations based on the collected trace. To answer RQ1, we use NodeAV to detect known atomicity violations on dataset-1 and find new atomicity violations on dataset-2. To answer RQ2, we measure the runtime overhead of NodeAV on dataset-2.

B. *Atomicity Violation Detection Results (RQ1)*

Detect known atomicity violations. We use dataset-1 to evaluate NodeAV's ability to detect known atomicity violations. Table III shows the detection result. The column *ID* refers to the ID in Table I. The column *Detected* indicates whether the known atomicity violation is detected, with value *Y* indicating yes and *N* indicating no. The column *#New violation* represents the number of new atomicity violations detected by NodeAV. We manually inspect the code to validate whether each new detected atomicity violation is real.

As shown in Table III, NodeAV detects 7 known atomicity violation out of 9. We further detect 8 new atomicity violations from 3 applications. After manual inspection, we find that all these newly detected atomicity violations are real. For example, in project *wikimedia/change-propagation*, NodeAV reveals a new buggy interleaving, which has not been found in the original bug report. This application is the prototype of our example in Fig. 2. The new atomicity violation happens when the second topic change overlaps the message of the first change between the processing of the first topic change, as shown in Fig. 3c. Note that, for atomicity violations with ID 4 and 5, NodeAV detects different numbers of new atomicity violations for the same application, since we use different test cases for these two known atomicity violations.

As shown in Table III, 2 known atomicity violations are not detected by NodeAV, i.e., the atomicity violations with ID 5 and 6. We further investigate why NodeAV misses these two atomicity violations. For the atomicity violation with ID 5, its related events are not triggered in the collected trace, and thus NodeAV cannot detect the atomicity violation from the unexplored execution trace. For the atomicity violation with ID

TABLE IV EXPERIMENT RESULT ON DATASET-2

ID	Project	#Violation	#FP	#Event	Time (s)				Tracing over-head (X)
					Orig	Trace	Analysis	Total	
1	michaelwittig/node-logger-file	2	0	1,752	19.4	31.9	31.2	63.1	1.6
2	substack/node-mkdir-p	0	0	23	4.3	5.5	0.1	5.6	1.3
3	hapijs/nes	1	0	34,211	4.7	7.5	3.7	11.2	1.6
4	telefonicaid/fiware-pep-steelskin	6	1	39,757	7.3	14.4	35.4	49.8	2.0
5	node-modules/agentkeepalive	0	0	10,873	65.7	70.1	0.6	70.7	1.1
6	nickewing/line-reader	1	0	633	0.4	0.7	0.4	1.1	1.8
7	sitegui/nodejs-websocket	1	0	676	0.2	0.9	1.0	1.9	4.5
8	JoshuaWise/better-sqlite3	0	0	9,346	12.1	12.9	0.8	13.7	1.1
9	derbyjs/racer	0	0	2,065	0.2	0.5	4.0	4.5	2.5
10	simplecrawler/simplecrawler	1	0	446	4.7	26.6	0.3	26.9	5.7

6, the accesses to the racy shared resource *socket* are not captured by NodeAV. It is because we do not model the Net API *destroy()* [32] in Section III.A, which destroys a socket.

Detect new atomicity violations. We use dataset-2 to evaluate NodeAV's ability to detect new atomicity violations. Table IV shows the result. The column *Project* denotes the applications under detection. The column *#Violation* represents the number of atomicity violations detected by NodeAV. The column *#FP* represents the number of false positives. We determine whether a reported atomicity violation is a false positive by manually reproducing it or reviewing the code.

As shown in Table IV, NodeAV detects 12 new atomicity violations from 6 applications. We manually validate whether these detected atomicity violations are true. Finally, we classify the detected atomicity violations into three categories: harmful atomicity violations, benign atomicity violations and false positives. We discuss them as follows.

NodeAV detects 3 harmful atomicity violations, 2 in *michaelwitting/node-logger-file* and 1 in *telefonicaid/fiware-pep-steelskin*. For example, in *michaelwittig/node-logger-file*, Event *create* creates a logging file, and the following event *log* writes logs into the logging file. However, another event *close*, which closes the logging file can occur between event *create* and *log*. As a result, event *log* throws an exception and fails. We have reported these atomicity violations on GitHub [33][34][35].

NodeAV detects 8 benign atomicity violations. Benign atomicity violations happen in two scenarios. First, the atomicity violation has no impact on the result. For example, the pattern Write-Read-Write in Fig. 7d indicates that event *t* gets dirty data. However, if event *e* and *u* store the same data, event *t* gets the same data in all schedules. Second, the impact resulting from the violation is expected by developers. For example, in the pattern Write-Write-Read in Fig. 7c, if event *t* happens between event *e* and *u*, the program checks that event *u* gets data written by event *t*, and throws an exception, which may be the expected behavior.

One atomicity violation reported in *telefonicaid/fiware-pep-steelskin* is false positive. NodeAV can explore an atomicity violation $e \longrightarrow t \longrightarrow u$, as shown Fig. 7b. However, event *t* modifies the application's state, resulting in that event *u* never be triggered. Thus, NodeAV reports a false positive.

C. Performance Overhead (RQ2)

We evaluate the performance of NodeAV on dataset-2. As shown in Table IV, the column *#Event* represents the number of events that NodeAV collects. The category *Time* shows the results about performance. The column *Orig* shows the time running the test suites on the original application, the column *Trace* shows the time running the test suites on the instrumented application to collect execution traces. The column *Analysis* shows the time for offline detection of atomicity violations based on the collected traces. The column *Total* shows the total time, calculated by *Trace+Analysis*.

As shown in Table II, our experimental applications are usually huge, including thousands of lines of code. Table IV shows that, during the trace collection, NodeAV obtained thousands of events, e.g., more than 39,000 events in the application *telefonicaid/fiware-pep-steelskin*. The column *Total* shows that NodeAV can analyze all these applications in no more than 71 seconds. The column *Tracing overhead* indicates the overhead caused by NodeAV when collecting traces, calculated by *Trace/Orig*. We can see that the runtime overhead ranges from 1.1X to 5.7X. This is acceptable for a dynamic analysis. The above results show that NodeAV can work on real-world Node.js applications.

VI. DISCUSSION

While our experiments show that NodeAV is promising in detecting atomicity violations in Node.js applications, we discuss potential threats and limitations in our approach.

A. Threats to Validity

Representativeness of our studied subjects. We select a number of atomicity violations in Node.js applications in our experiments. First, our selected atomicity violations come from real-world Node.js applications. Second, these atomicity violations have been used in existing studies, e.g., NodeCB [8] and Node.fz [9]. Thus, we believe our studied atomicity violations represent the real-world bugs. Most of our selected Node.js applications are popular, and have lots of stars in GitHub. We believe that these applications are representative.

B. Limitations

Supporting more Node.js versions. We utilize the *async_hooks* module to trace event scheduling operations in Node.js. The *async_hooks* module is only supported by Node.js 8.6 or above. For now, we cannot handle Node.js applications that run on older Node.js versions yet. A new approach that can analyze these applications on older Node.js versions is required.

API modeling. In our approach, we abstract each file system API into one or multiple operations with *read* and *write* types.

That's said, we only consider equivalent influence on the file when calling an API but ignore the state of a file. We also treat asynchronous I/O as synchronous operations. A more precise model about file systems, e.g., the state machine in 2ndStrike [36] can be integrated into NodeAV. In addition, modeling more APIs, such as third-party database and network APIs, can also be integrated into NodeAV, and further improves the ability of NodeAV.

VII. Related Work

To the best of our knowledge, no previous work presents (1) a formal definition of atomicity violations, (2) a happens-before relation and (3) an atomicity violation tool, for the event-driven Node.js applications. The event-driven architecture has been widely used in various software platforms, e.g., Android, client-side and server-side JavaScript applications. Some approaches that target concurrency bugs in these systems has been proposed. In this section, we discuss related work that is close to ours.

Multithreaded programs. Concurrency bug detection is a well-studied topic in multithreaded programs. Many approaches and tools have been developed to identify concurrency bugs in multithreaded programs, e.g., FastTrack [37], Eraser [38] for data races, AVIO [10], CTrigger [11], and Atomizer [39] for atomicity violations. These approaches are usually based on a multithreaded programming model, and use multithreaded specific happens-before relation to detect data races or atomicity violations. The event-driven model in Node.js applications is different from multithreaded programming model. For example, atomicity violations in multithreaded programs concern whether several shared variable accesses in a thread are interrupted by another thread, while atomicity violations in event-driven Node.js applications concern whether two consecutive events are interrupted by another event. Therefore, our work is orthogonal to existing studies in multithreaded programs.

Android applications. Android adopts a mixed event-driven and multithreaded programming model. Recently, some approaches have been proposed to detect event races in Android applications. CAFA [14], DroidRacer [15] and EventTrack [40] use dynamic analysis approaches to detect event races in Android applications. They usually build happens-before relation among events and threads based on the mixed programming model. SIERRA [41] proposes a precise and scalable static approach for detecting Android event races. Node.js adopts different event-driven model from Android applications, e.g., event queues and event scheduling. Further, existing studies mainly concern event races, which involve two events. While, our work mainly concerns atomicity violations among a group of events.

Client-side JavaScript applications. Concurrency bugs in client-side JavaScript applications have drawn researchers' attention recently. WAVE [16], WebRacer [17] and EventRacer [42] present the happens-before relation for client-side JavaScript applications, and further detect races in them. The happens-before model in client-side JavaScript applications includes DOM, web page loading, etc. ARROW [43] further proposes to automatically fix races in client-side JavaScript applications. Node.js applications do not concern DOM and web page loading that are the key elements in client-side JavaScript applications. Further, Node.js adopts different event queue model from client-side JavaScript applications. Thus, the above approaches cannot be directly applied on Node.js applications. In addition, existing studies on client-side JavaScript applications cannot handle atomicity violations among events.

Node.js applications. The Node.js platform is relatively new, and as a result, there is limited work on analysis tools for Node.js applications, especially on concurrency bug detection. NodeCB [8] studies 57 concurrency bugs in real-world Node.js applications and obtains some interesting findings, e.g., 65% of concurrency bugs in Node.js are atomicity violations. This study motivates us to design an effective tool to detect atomicity violations in Node.js. Node.fz [9] is an event fuzzing tool for Node.js applications. It randomly shuffles the enqueue order in the event queues, and tries to yield different schedules and increase the possibility of race condition. While, our work systematically validates atomicity violations based on an execution trace. GEMs [44] presents a new parallel programming abstraction in Node.js, which combines message passing and shared-memory parallelism. Madsen et al. [45] proposes a static analysis approach to build the event-based call graph and detect bugs related to event handling, e.g., unhandled events and listeners registered too late. Loring et al. [28] proposes the first formalization of asynchronous execution model in Node.js, and async-contexts to formalize relationships between asynchronous events. The above studies are orthogonal to ours.

As a new platform, other issues in Node.js have also drawn researchers' attention. Since Node.js heavily depends on event handlers to achieve effective responsiveness, if an event handler performs a heavy computation, then it can block the event loop, and cause denial of service attacks [46]. Ojamaa et al. [47] argue how the Node.js design affects the security of Node.js applications. Synode [48] proposes a static analysis approach to prevent injection attack.

VIII. Conclusion

Although each event is guaranteed to be processed atomically, no effective mechanism in Node.js is provided to maintain the atomicity of an atomic event group, in which the events should be processed without interruption. This kind of atomicity violations are common in Node.js applications and often cause serious consequences. In this paper, we propose NodeAV to predictively detect atomicity violations based on a collected execution trace. We build the precise happens-before relation for events in the execution trace, and infer events that should be executed together without interruption. We further propose several atomicity violation patterns in the context of event-driven Node.js applications. We evaluate NodeAV on real-world Node.js applications and the experimental results show NodeAV can detect known atomicity violation, as well as new atomicity violations.

IX. Acknowledge

This work was partially supported by National Key R&D Program of China (2017YFB1001804), National Natural Science Foundation of China (61732019, 61672506, 61702490), Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036), and Youth Innovation Promotion Association at Chinese Academy of Sciences.

REFERENCES

[1] "Node.js Foundation." [Online]. Available: https://nodejs.org/en/.

[2] "Tracking the Explosive Growth of Open-Source Software." [Online]. Available: https://techcrunch.com/2017/04/07/tracking-the-explosive-growth-of-open-source-software/.

[3] "The npm Repository." [Online]. Available: https://www.npmjs.com/.

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

[5] "Node.js in LinkedIn." [Online]. Available: https://venturebeat.com/2011/08/16/linkedin-node/.

[6] "Node.js in Yahoo." [Online]. Available: https://yahooeng.tumblr.com/node.

[7] "Node.js in Mozila." [Online]. Available: https://medium.com/mozilla-tech/mozilla-and-node-js-33c13e29beb1.

[8] J. Wang *et al.*, "A Comprehensive Study on Real World Concurrency Bugs in Node.js," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 520–531.

[9] J. Davis, A. Thekumparampil, and D. Lee, "Node. fz: Fuzzing the Server-Side Event-Driven Architecture," in *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, 2017, pp. 145–160.

[10] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, pp. 37–48.

[11] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009, pp. 25–36.

[12] B. Lucia, J. Devietti, K. Strauss, and L. Ceze, "Atom-Aid: Detecting and Surviving Atomicity Violations," in *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, 2008, pp. 277–288.

[13] G. Upadhyaya, S. P. Midkiff, and V. S. Pai, "Automatic Atomic Region Identification in Shared Memory SPMD Programs," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2010, pp. 652–670.

[14] C.-H. Hsiao *et al.*, "Race Detection for Event-Driven Mobile Applications," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014, pp. 326–336.

[15] P. Maiya, A. Kanade, and R. Majumdar, "Race Detection For Android Applications," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014, pp. 316–325.

[16] S. Hong, Y. Park, and M. Kim, "Detecting Concurrency Errors in Client-Side Java Script Web Applications," in *Proceedings of the 17th International Conference on Software Testing, Verification and Validation (ICST)*, 2014, pp. 61–70.

[17] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby, "Race Detection for Web Applications," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012, pp. 251–262.

[18] G. Safi, A. Shahbazian, W. G. J. Halfond, and N. Medvidovic, "Detecting Event Anomalies in Event-Based Systems," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015, pp. 25–37.

[19] S. Artzi, J. Dolby, S. H. Jensen, A. Moller, and F. Tip, "A Framework for Automated Testing of JavaScript Web Applications," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 571–580.

[20] P. Bielik, V. Raychev, and M. Vechev, "Scalable Race Detection for Android Applications," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015, pp. 332–348.

[21] Y. Hu, I. Neamtiu, and A. Alavi, "Automatically Verifying and Reproducing Event-Based Races in Android Apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, 2016, pp. 377–388.

[22] "Seven Event Queues in Node.js." [Online]. Available: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/.

[23] "A Node.js Application: Change Propagation." [Online]. Available: https://github.com/wikimedia/change-propagation.

[24] "No.84 Issue in Change Propagation." [Online]. Available: https://github.com/wikimedia/change-propagation/pull/84.

[25] "Promises." [Online]. Available: https://www.promisejs.org/.

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

[27] "Standard Built-in Objects." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects.

[28] M. C. Loring, M. Marron, and D. Leijen, "Semantics of Asynchronous JavaScript," in *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages (DLS)*, 2017, pp. 51–62.

[29] "Async Hooks." [Online]. Available: https://nodejs.org/api/async_hooks.html.

[30] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2013, pp. 488–498.

[31] "Cloc: count lines of code." [Online]. Available: https://github.com/AlDanial/cloc.

[32] "Net - Node.js v11.7.0 Documentation." [Online]. Available: https://nodejs.org/api/net.html#net_socket_destroy_exception.

[33] "A race condition _log() while closing the endpoint." [Online]. Available: https://github.com/michaelwittig/node-logger-file/issues/5.

[34] "A race condtion logging a file while rolling a file." [Online]. Available: https://github.com/michaelwittig/node-logger-file/issues/4.

[35] "A race condition invalidateToken while authenticating a request." [Online]. Available: https://github.com/telefonicaid/fiware-pep-steelskin/issues/412.

[36] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin, "2ndStrike: Toward Manifesting Hidden Concurrency Typestate Bugs," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011, pp. 239–250.

[37] C. Flanagan and S. N. Freund, "FastTrack: Efficient and Precise Dynamic Race Detection," *Acm Sigplan Not.*, vol. 44, no. 6, pp. 121–133, 2009.

[38] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.

[39] C. Flanagan and S. N. Freund, "Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2004, pp. 256–267.

[40] P. Maiya and A. Kanade, "Efficient Computation of Happens-Before Relation for Event-Driven Programs," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2017, pp. 102–112.

[41] Y. Hu and I. Neamtiu, "Static Detection of Event-based Races in Android Apps," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018, pp. 257–270.

[42] V. Raychev, M. Vechev, and M. Sridharan, "Effective Race Detection for Event-Driven Programs," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages, & Applications (OOPSLA)*, 2013, pp. 151–166.

[43] 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 the 25th International Symposium on Software Testing and Analysis (ISSTA)*, 2016, pp. 201–212.

[44] 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, & Applications (OOPSLA)*, 2016, pp. 531–547.

[45] M. Madsen, F. Tip, and O. Lhoták, “Static Analysis of Event-Driven Node.js JavaScript Applications,” *Acm Sigplan Not.*, vol. 50, no. 10, pp. 505–519, 2015.

[46] J. Davis, G. Kildow, and D. Lee, “The Case of the Poisoned Event Handler: Weaknesses in the Node.js Event-Driven Architecture,” in *Proceedings of the European Workshop on Systems Security (EuroSec)*, 2017, pp. 1–6.

[47] A. Ojamaa and K. Düüna, “Assessing the Security of Node. js Platform,” in *Proceedings of the International Conference for Internet Technology and Secured Transactions (ICITST)*, 2012, pp. 348–355.

[48] C.-A. Staicu, M. Pradel, and B. Livshits, “Understanding and Automatically Preventing Injection Attacks on Node.js,” in *Proceedings of Network and Distributed Systems Security (NDSS)*, 2018.