Static Analysis of Event-Driven Node.js JavaScript Applications

Magnus Madsen

University of Waterloo
mmadsen@uwaterloo.ca

Frank Tip

Samsung Research America
ftip@samsung.com

Ondřej Lhoták

University of Waterloo
olhotak@uwaterloo.ca

Abstract

Many JavaScript programs are written in an event-driven style. In particular, in server-side Node.js applications, operations involving sockets, streams, and files are typically performed in an asynchronous manner, where the execution of listeners is triggered by events. Several types of programming errors are specific to such event-based programs (e.g., unhandled events, and listeners that are registered too late). We present the *event-based call graph*, a program representation that can be used to detect bugs related to event handling. We have designed and implemented three analyses for constructing event-based call graphs. Our results show that these analyses are capable of detecting problems reported on StackOverflow. Moreover, we show that the number of false positives reported by the analysis on a suite of small Node.js applications is manageable.

Categories and Subject Descriptors   F3.2 [*Semantics of Programming Languages*]: Program Analysis

Keywords   event based-systems; static analysis; JavaScript

1.   Introduction

JavaScript has rapidly become one of the most popular programming languages[1] and is now being used in several areas beyond its original domain of client-side scripting. For example, Node.js[2] is a popular platform for building server-side web applications written in JavaScript, and JavaScript is one of the primary languages used for application development in the Tizen Operating System[3] for mobile devices. As a testament to the popularity of Node.js, the `npm` pack-

---

[1] See `http://langpop.com/`.

[2] See `http://www.nodejs.org/`.

[3] See `https://www.tizen.org/`.

age repository has recently surpassed Maven Central and RubyGems with more than 132,000 available packages and over 250 packages added every day.

As a result of JavaScript's increasing popularity, there has been a growing demand for tools that assist programmers with tasks such as program understanding and maintenance [23, 31], bug detection and localization [3, 17], refactoring [6, 7], and detecting and preventing security vulnerabilities [9, 26, 33]. Many such tools rely on static analysis to approximate a program's behavior. One program representation that is commonly used in static analysis is the call graph, which associates with each call site in a program the set of functions that may be invoked from that site. For example, a tool for finding security vulnerabilities might use a call graph to detect possible data flows from tainted inputs to security-sensitive operations, and a refactoring tool may rely on a call graph to determine how to inline a function call.

However, a traditional call graph reflects only the inter-procedural flow of control due to function calls, and ignores the event-driven flow of control in many JavaScript applications. For example, interactive applications that access the HTML Document Object Model (DOM) typically do so by associating event listeners with DOM nodes that correspond to fragments of an HTML document in a browser. Similarly, server-side applications based on Node.js are typically written in an event-driven style which heavily relies on callbacks that are invoked when an asynchronously executed operation has completed. Several new types of bugs may arise in event-driven programs. For example, an application may emit events for which no listener has been registered yet, or an event listener may be unreachable code because an event name was mis-spelled or the event listener was registered on the wrong object. In Section 2, we discuss a few examples of such errors that were reported on StackOverflow[4], a popular discussion forum for programming-related problems.

Our goal in this paper is to detect errors in event-driven Node.js JavaScript programs using static analysis. We observe that traditional call graphs are not suitable in this context, because they do not reflect the flow of control that is implied by the registration of event listeners and the emission of events. Therefore, we propose the *event-based call*

---

[4] http://www.stackoverflow.com/

graph, an extension of the traditional notion of a call graph with nodes and edges that reflect the flow of control due to event handling.

We show how event-based call graphs can be used as the basis for detecting several types of errors, and present a family of analyses for computing event-based call graphs with varying cost and precision. Specifically, we extend the λ_{JS} calculus [10] with constructs for event listener registration and event emission. Based on this extended calculus, we present an analysis framework and three instantiations that we will refer to as the *baseline*, *event-sensitive* and *listener-sensitive* analysis. The event- and listener sensitive analyses compute separate dataflow facts based on what events have been emitted and what listeners are registered, respectively. We implement these variants in a static analysis tool called RADAR. We experimentally evaluate RADAR by applying it to buggy Node.js programs from the StackOverflow website and to programs from the Node.js documentation [5] and the *Node.js in Action* book [4].

In summary, our paper makes the following contributions:

- We identify errors that may arise in event-based JavaScript programs.

- We extend the λ_{JS} calculus with constructs for event listener registration and event emission.

- We define the notion of an *event-based call graph*, which extends the traditional notion of a call graph with nodes and edges that reflect the flow of control due to events, and show how this graph can be used to detect errors.

- We demonstrate that event-based call graphs are useful for finding errors by applying them to several buggy Node.js programs from the StackOverflow website.

- We present three analyses, with varying cost and precision, for constructing event-based call graphs. For convenience, we will refer to these as the *baseline*, *event-sensitive* and *listener-sensitive* analysis.

2. Motivating Examples

In this section, we examine a few examples of buggy Node.js programs (taken from StackOverflow) that reflect actual problems experienced by Node.js developers. Our goal is to develop a static analysis that can detect these bugs, pinpoint their location to the programmer, and report few to none spurious warnings. We chose to focus on these StackOverflow examples because they reflect real bugs that real programmers are struggling with. While they are small, they still reveal that existing techniques that assume a single event loop (e.g., TAJS [13] and JSAI [18]) are unable to avoid spurious warnings (on corrected programs).

2.1 StackOverflow Question 19167407

Consider the program of Figure 1. Here, the programmer's goal is to write an application that reads an `mp4` stream from YouTube, and write its contents to a local file. To this end,

```
1  var writing = fs.createWriteStream('video.mp4');
2  var stream =
3    ytdl('https://www.youtube.com/
4                watch?v=jofNR_WkoCE',
5        { filter: function(format) {
6                return format.container
7                       === 'mp4'; },
8          quality: "lowest" });
9  stream.pipe(writing);
10
11 var completed_len = 0;
12 var total_len = 0;
13
14 writing.on('data', function(chunk) {
15     console.log('received data!');
16     completed_len += chunk.length;
17 });
18
19 writing.on('close', function () {
20     console.log('close');
21     res.send('completed!');
22 });
```

Figure 1. A Node.js program in which the programmer registers a listener with the wrong stream. See `http://stackoverflow.com/questions/19167407/`.

the programmer relies on the `ytdl()` function provided by the YouTube downloader module for Node.js.

The program first creates a stream where the contents are to be written, on line 1. Then, on lines 2–8 a stream is created for reading the contents from YouTube. Next, on line 9, the contents of the latter stream are piped into the former. After initializing some counters on lines 11 and 12, two listeners are registered with the write-stream. First, on lines 14–17 a listener is associated with `data` events to update the log and counter. Second, on lines 19–22, a listener is associated with `close` events to write a message to the log, and create an appropriate response.

The programmer reports that "Weirdly enough, even though 'close' fires whenever the download is done, I'm not getting any logs from the 'data' event. The video is written correctly." The problem (correctly diagnosed on StackOverflow) is that `data` events are only associated with readable streams, and not with writable streams. On lines 14–17, the programmer inadvertently bound the event listener to the wrong stream.

Our static analysis tool can detect this type of "dead listener" bug and report that the `data` event is never emitted on the writable stream object.

2.2 StackOverflow Question 19081270

Figure 2 shows another problematic Node.js code fragment taken from StackOverflow, which relies on the `restlr` library to facilitate interaction with HTTP servers. Lines 4–8 assign a function to variable `restlerHtmlFile`. The call `rest.get(Url)` within this function creates a GET request to obtain the contents of a URL. The call `.on('complete',`

```
1 var fs = require('fs');
2 var rest = require('restler');
3
4 var restlerHtmlFile = function(url) {
5   rest.get(url).on('complete', function(res) {
6       fs.writeFileSync('file.html', res);
7   });
8 };
9
10 if (require.main == module) {
11   restlerHtmlFile('http://obscure-refuge-7370.
12                   herokuapp.com/');
13   fs.readFileSync('file.html');
14 } else {
15   exports.checkHtmlFile = checkHtmlFile;
16 }
```

Figure 2. A Node.js program in which the programmer is incorrectly combining synchronous and asynchronous calls. See `http://stackoverflow.com/questions/19081270/`.

...) on line 5 serves to write the page contents to `file.html` when the request completes. Then, on line 11, the function bound to `restlerHtmlFile` is invoked to read the contents of the URL `http://obscure-refuge-7370.herokuapp-.com/` into the file `file.html`, and on line 13, this file is read by calling `fs.readFileSync('file.html')`. The programmer reports on StackOverflow that the program crashes with an error message "no such file or directory 'file.html' at Object.fs.openSync (fs.js:427:18)".

The problem here has to do with the fact that the function passed in the call `.on('complete', ...)` on line 5 is invoked *asynchronously*, when the GET request has completed. However, line 13, in which the generated file `file.html` is written, executes immediately after line 11, without making sure that the file creation has completed.

There are various ways in which the code can be fixed. One solution, which is suggested on StackOverflow, is to move the call `fs.readFileSync('file.html')` inside the definition of the asynchronous event listener, so that it will not execute before the writing of the file has completed.

The question at this point is how the programmer could have observed that the code is buggy. Here, the key issue is that the programmer implicitly assumed that the read-operation on line 13 will always execute after the write-operation on line 6. The static analyses that we present in this paper can determine that no such ordering exists in this case, and can be used in the context of a bug-finding tool to alert programmers early to this kind of problem.

2.3 Limitations of Current Static Analyses

Figure 3 is an example taken from the online Node.js documentation that shows how to correctly sequence asynchronous operations. The top part of the program performs `rename` and `stat` operations, but although the `rename` call appears before the `stat` call, there is no guarantee that the

```
1 // incorrect sequencing
2 fs.rename('a.txt', 'b.txt');
3 fs.stat('b.txt', function (err, stats) {
4     console.log('size: ' + stats.size);
5 });
```

```
1 // correct sequencing
2 fs.rename('a.txt', 'b.txt', function () {
3   fs.stat('b.txt', function (err, stats) {
4       console.log('size: ' + stats.size);
5   });
6 });
```

Figure 3. An example of incorrect (top) and correct (bottom) sequencing of asynchronous operations in Node.js.

two operations will be executed in that order. The bottom part shows the corrected program where the `stat` call is nested inside a callback for the `rename` operation.

Current state-of-the-art static analyzers for JavaScript such as TAJS [13] and JSAI [18] model events in a way that allows them to detect the error in the top program. However, when these tools are applied to the corrected program at the bottom they will still produce a spurious warning! This is because TAJS and JSAI use a simple event model where the event loop is implemented as "... a non-deterministic execution of event-handling functions." [18]. However, in order to accurately capture the behaviour of the program in the bottom part of Figure 3 it is crucial that the analysis can prove that the outer callback is *always* executed before the inner callback. That is, if there is only a single non-deterministic event-loop then the analysis cannot rule out the spurious behaviour that the `stat` may happen before the `rename`. In this paper we develop static analyses that can rule out such behaviour by an appropriate unfolding of the event loop.

3. Language

The focus of this paper is to develop program representations and analyses that are useful for understanding the behavior of event-based JavaScript programs. JavaScript is a complex language with features such as prototype-based inheritance, dynamic property access, implicit coercions and on-the-fly code evaluation with `eval`. Much recent research effort has been invested in studying these features [7, 13, 14, 18, 21, 25]. However, surprisingly little research has focused on the event-driven nature of control flow in JavaScript applications. We wish to provide a formalization of how events interact with JavaScript, but without having to deal with the complicated features mentioned above. Therefore, we extend a minimal JavaScript calculus [10] with constructs for event handling.

3.1 Design Choices

We highlight some important design choices that were made when formalizing the semantics of events and listeners.

These choices reflect similar choices made by the Node.js developers and of JavaScript in general.

- Execution is single-threaded and non-preemptive. An event listener must run to completion before another event listener may begin execution.

- An object may have multiple event listeners registered for the same event. The event listeners are executed in registration order, if that event is emitted.

- If the execution of an event listener f for an event τ registers an event listener f' for the same event τ then f' is not executed until τ is emitted again.

- Event names are drawn from a known finite set.[5]

3.2 Syntax of λ_ϵ

Guha et al. describe a core calculus for JavaScript called λ_{JS} [10]. The syntax of this language is shown in Figure 4. The reduction semantics are provided in their paper and are largely unchanged, except as noted below. The language λ_{JS} has primitive values, objects, functions and references. The primitive values are booleans, numbers, strings, `null` and `undefined`. Values are primitive values, heap locations, objects or functions. Objects are immutable maps from fields to values. Functions are lambda expressions. The expressions are standard except for two operations that manipulate the heap: The `ref` expression evaluates its argument to a value, stores that value in memory and returns the address of where it was stored. The `deref` expression evaluates its argument to an address and retrieves the value of that address from memory. JavaScript features that are not in λ_{JS} are compilable from JavaScript into λ_{JS} as shown in [10]. Thus λ_{JS} provides a minimal calculus which is still sufficiently expressive to model all aspects of JavaScript (except the `eval` expression).

We turn λ_{JS} into an event-based language λ_ϵ by introducing *events* and *listeners* into the syntax and semantics. Specifically, we introduce three new terms:

Listen: $e_1.\texttt{listen}(\tau, e_2)$: An expression that registers a *listener* for when τ is *emitted* on the receiver object. The expression e_1 is the address of the receiver object on which the listener is registered, the expression e_2 is the event listener (i.e., a function value), and τ is the *event name*.

Emit: $e_1.\texttt{emit}(\tau, e_2)$: Emits (i.e., triggers) an event on an object, which results in *scheduling* all event listeners *registered* for that event on that object. If multiple event listeners are registered for the same event then they are *scheduled* in registration order. The expression e_1 is the address of the receiver object, τ is the event name and e_2 is the argument passed to the event listener(s).

[5] This assumption is only made to simplify the presentation of the semantics rules. In practice, event names are ordinary strings that may be computed at run-time. Our implementation, described in Section 6, handles such cases by using a string analysis to track computed event names.

$$
\begin{array}{llll}
c \in \mathit{Cst} & = & \mathit{bool} \mid \mathit{num} \mid \mathit{str} \mid \texttt{null} \mid \texttt{undef} & \text{[constant]}
\end{array}
$$

$$
\begin{array}{llll}
v \in \mathit{Val} & = & c & \text{[literal]} \\
& \mid & a & \text{[address]} \\
& \mid & \{\mathit{str} : v \cdots\} & \text{[object]} \\
& \mid & \lambda(x \cdots)\ e & \text{[function]}
\end{array}
$$

$$
\begin{array}{llll}
e \in \mathit{Exp} & = & v & \text{[value]} \\
& \mid & x & \text{[variable]} \\
& \mid & e = e & \text{[assignment]} \\
& \mid & \texttt{let}\ (x = e)\ e & \text{[binding]} \\
& \mid & e\,(e \cdots) & \text{[call]} \\
& \mid & e.f & \text{[field load]} \\
& \mid & e.f = e & \text{[field store]} \\
& \mid & \texttt{ref}\ e & \text{[address of]} \\
& \mid & \texttt{deref}\ e & \text{[value at]}
\end{array}
$$

$$
\begin{array}{lll}
x \in \mathit{Var} & = & \text{is a finite set of variable names.} \\
f \in \mathit{Fld} & = & \text{is a finite set of field names.} \\
a \in \mathit{Addr} & = & \text{is an infinite set of memory addresses.} \\
\lambda \in \mathit{Lam} & = & \text{is the set of all lambda expressions.}
\end{array}
$$

Figure 4. Syntax of λ_{JS}.

$$
\begin{array}{llll}
e \in \mathit{Exp} & = & \bullet & \text{[event loop]} \\
& \mid & e.\texttt{listen}(\tau, e) & \text{[attach listener]} \\
& \mid & e.\texttt{emit}(\tau, e) & \text{[emit event]}
\end{array}
$$

$$
\begin{array}{lll}
\tau \in \mathit{Event} & = & \text{is a finite set of event names.}
\end{array}
$$

Figure 5. Syntax of λ_ϵ.

$$
\begin{array}{lll}
E & = & \square \\
& \mid & E.\texttt{listen}(\tau, e) \mid v.\texttt{listen}(\tau, E) \\
& \mid & E.\texttt{emit}(\tau, e) \mid v.\texttt{emit}(\tau, E)
\end{array}
$$

Figure 6. Evaluation Contexts for λ_ϵ. The $\square$ symbol represents the hole in the evaluation context.

Loop: $\bullet$: An expression which represents the "event-loop", i.e., the situation when the call stack is empty and scheduled event listeners are extracted from the event queue and executed. We assume, by transformation if necessary, that the last expression of the "top-level" function is a single $\bullet$. Intuitively, the purpose of $\bullet$ is to explicitly represent when event listeners are allowed to execute.

We call the extended language λ_ϵ. The syntax of the new terms is shown in Figure 5.

$$\begin{aligned}
\sigma \in \textit{Heap} &= \textit{Addr} \to \textit{Val} \\
\vartheta \in \textit{Listener} &= \textit{Addr} \times \textit{Event} \to \textit{Lam}^\star \\
\pi \in \textit{Queue} &= \textit{Addr} \times \textit{Event} \to (\textit{Lam} \times \textit{Val})^\star \\
s \in \textit{State} &= \textit{Heap} \times \textit{Listener} \times \textit{Queue} \times \textit{Exp}
\end{aligned}$$

Figure 7. Runtime of λ_ϵ.

$$\frac{\langle \sigma, \vartheta, \pi, e \rangle \hookrightarrow^\star \langle \sigma', \vartheta', \pi', e' \rangle}{\langle \sigma, \vartheta, \pi, E[e] \rangle \hookrightarrow \langle \sigma', \vartheta', \pi', E[e'] \rangle} \quad \text{[E-CTX]}$$

$$\frac{\begin{array}{c} o \in \textit{Addr} \quad f = \lambda\,(x \cdots)\ e \\ \vartheta' = \vartheta[(o, \tau) \mapsto f :: \vartheta(o, \tau)] \end{array}}{\langle \sigma, \vartheta, \pi, o.\mathtt{listen}(\tau, f) \rangle \hookrightarrow \langle \sigma, \vartheta', \pi, \mathtt{undef} \rangle} \quad \text{[E-LISTEN]}$$

$$\frac{\begin{array}{c} o \in \textit{Addr} \quad \vartheta(o, \tau) = \lambda_1 :: \cdots :: \lambda_n \\ \pi' = \pi[o \mapsto (\lambda_1, v) :: \cdots :: (\lambda_n, v) :: \pi(v)] \end{array}}{\langle \sigma, \vartheta, \pi, o.\mathtt{emit}(\tau, v) \rangle \hookrightarrow \langle \sigma, \vartheta, \pi', \mathtt{undef} \rangle} \quad \text{[E-EMIT]}$$

$$\frac{\begin{array}{c} o \in \textit{Addr} \quad \pi(o, \tau) = (\lambda_1, v_1) :: \cdots :: (\lambda_n, v_n) \\ e = \lambda_n(v_n); \ldots ; \lambda_1(v_1) \quad \pi' = \pi[(o, \tau) \mapsto \textit{Nil}] \end{array}}{\langle \sigma, \vartheta, \pi, \bullet \rangle \hookrightarrow \langle \sigma, \vartheta, \pi', e; \bullet \rangle} \quad \text{[E-LOOP]}$$

Figure 8. Semantics of λ_ϵ. Here the notation $x :: xs$ is cons and *Nil* denotes the empty list.

3.3 Runtime of λ_ϵ

The runtime of λ_ϵ consists of (1) a heap σ, which is a partial map from addresses to values, (2) a map of registered event listeners ϑ, which is a map from addresses and event names to a list of event listeners, and (3) a queue of scheduled functions π, which is also a map from addresses and event names to an ordered list of functions calls. (The calls are not *evaluated* until they are executed.) The runtime environment of λ_ϵ is depicted in Figure 7. The queue π must maintain a separate list of listeners for each address and event name because no total ordering exists on the event listener executions. However, for each object and event, the order of listener executions is defined, so a single global unordered set of pending listener executions would be imprecise. A configuration is a 4-tuple $c = \langle \sigma, \vartheta, \pi, e \rangle$ consisting of the heap, the event listeners, the queue and the current expression.

3.4 Semantics of λ_ϵ

The semantics of λ_ϵ is defined by adding four new rules to those in [10]. As in [10], we use *evaluation contexts* [8] to specify the order in which sub-expressions are evaluated in expressions.

An evaluation context is a tree with a hole. The hole is denoted by $\square$ and represents where in the expression evaluation should proceed. The evaluation contexts for the new constructs are shown in Figure 6. The semantics for the new terms are in Figure 8 and explained further below:

[E-CTX] : This rule uses the evaluation context to evaluate sub-terms and re-compose them. That is, if some term e_1 can evaluate to e_2 in one or more steps, and e_1 occurs in the hole according to the evaluation context, then we can plug the hole with e_2. This rule is adapted from [10].

[E-LISTEN] $o.\mathtt{listen}(\tau, f)$: Registers f as an event listener for event τ on the object pointed-to by o, if o is an address and f is a function value.

[E-EMIT] $o.\mathtt{emit}(\tau, v)$: Adds all event listeners which are registered for event τ on the object pointed-to by o to the event queue. The v value is used as argument to each of the event listeners.

[E-LOOP] $\bullet$: Non-deterministically chooses an address o and an event τ and executes all event listeners scheduled for that object and event by moving them from the queue into the current expression as a sequence of statements. Notice that listeners are executed in registration order (since π is maintained in reverse-registration order).

3.5 Other Event Features

Asynchronous Callbacks. Asynchronous callbacks, as shown in Figure 3, can be expressed in terms of the primitives that we have defined for λ_ϵ. To express a function f that asynchronously executes another function g, we make f allocate a fresh object, register function g on that object as a listener for some event, and emit that event on the object:

```
1 function f(g) {
2   var x = new Object(); // fresh object
3   x.listen("e", g); // register g for event e
4   x.emit("e"); // emit "e" on the object
5 }
```

Listener Unregistration. Node.js supports unregistration of event listeners. However, this feature is used rarely because event listeners typically follow the lifetime of an object by being registered immediately after the object is created and then never removed. In all code we have looked at, we have not yet encountered a situation where unregistering listeners was needed. If necessary, this could be supported by a straightforward extension to the λ_ϵ semantics.

4. Beyond Call Graphs

A (traditional) call graph is a directed graph that connects *call sites* with *call targets* (i.e., function declarations). Call graphs are useful for debugging, refactoring, and many other applications. In languages with polymorphism and/or higher-order functions, the call graph is not immediately available from the source code, but must be statically approximated, e.g., using points-to analysis. Traditionally, a call graph helps the programmer answer two key questions:

- Who are the *callers* of a function? (i.e., "who calls me")

- Who are the *callees* of a function? (i.e., "who do I call?")

However, in languages with asynchronous callbacks and event listeners a traditional call graph provides incomplete information because it does not reflect precisely how events give rise to indirect calls.

In a traditional call graph, the callers of function λ are the functions that may invoke it. However, in event-based systems, control flow is also determined by: (i) functions that may *register* a function λ as a listener, and (ii) functions that may *emit* an event that causes λ to be scheduled for execution. In traditional call graphs, this correlation between events and listeners is lost, and functions that are invoked due to event-handling are typically modeled as "roots" of a call graph (i.e., call-backs from a runtime environment).

4.1 Event-Based Call Graphs

We overcome the limitations of traditional call graphs by introducing the *event-based call graph*, a directed graph whose nodes are expressions and lambdas from the source code, and which has four kinds of edges. In Section 5 we present three analyses to compute an over-approximation of the event-based call graph.

A *call edge* connects a call expression to a function declaration λ:

$$\{e\,(e\cdots) \rightarrow \lambda \mid \text{if the call may invoke } \lambda\}$$

As stated earlier, call edges answer the questions: (i) which functions directly invoke a function? and (ii) which functions are directly invoked by a function? A *listen edge* connects a listen expression to a function λ:

$$\{e_1.\texttt{listen}(\tau, e_2) \xrightarrow{\tau} \lambda \mid \quad \text{if the } \texttt{listen}\text{-expression } \}$$
$$\text{may register function } \lambda$$

From such listen edges, the following questions can be answered: (i) Which functions does a given listen expression register as listeners for a given event? (ii) Which listen expressions may register a given function λ as a listener for an event? An *emit edge* connects an emit expression to a function declaration λ:

$$\{e.\texttt{emit}(\tau) \xrightarrow{\tau} \lambda \mid \text{if event } \tau \text{ may schedule } \lambda\}$$

From the emit edges in an event-based call graph, the following questions can be answered: (i) which functions does an emit expression schedule for execution by emitting the event τ? and (ii) which emit expressions may cause a given function to be scheduled for execution?

Finally, we introduce *may-happen-before* edges between expressions:

$$\{(e_1, e_2) \mid \text{if } e_1 \text{ may happen before } e_2.\}$$

where e_1 and e_2 are either `listen` or `emit` expressions. Here, e_1 may possibly execute before e_2 *only if there is an*

```
1  let (x = {})
2    let (y = {})
3      let (f = λ₀() x.listen(τ₃, λ₃ () ...))
4        x.listen(τ₁, λ₁() y.emit(τ₂));
5        y.listen(τ₂, λ₂() f());
6        x.emit(τ₁)
```

Figure 9. A small λ_ϵ program. The program contains three event listeners λ_1, λ_2 and λ_3 which are registered for the events τ_1, τ_2 and τ_3, respectively.

edge (e_1, e_2). If, at the same time, there is the edge (e_2, e_1) then e_1 and e_2 may execute in any order. Crucially, if there is no (e_1, e_2) edge then e_1 *cannot be executed before* e_2. Thus, the may-happen-before relationship reflects a temporal relationship between event listener registration and event emission.

We can use these edges to decide whether an event listener can be executed. Specifically, in order for an event listener λ to be executed, three edges must be present:

1. There must be a listen edge, for an event τ, from a `listen` expression to λ.

2. There must be an emit edge, for an event τ, from an `emit` expression to λ.

3. A may-happen-before edge must exist from the `listen` expression to the `emit` expression.

If any of these three edges is missing, for any listener λ, then that listener can never be executed by the event system. Note that the above definition implicitly requires that the `emit` and `listen` expressions themselves can be executed, otherwise they would not participate in a may-happen-before relation.

Example. Figure 9 shows a small program with calls, listen and emit expressions. Figure 10 shows the event-based call graph for this program. It has a single *call* edge from the call site `f()` on line 5 to λ_0 on line 3. It has two *emit* edges: one from the emit on line 4 to λ_2 on line 5 and one from the emit on line 6 to λ_1 on line 4. It has three *listen* edges: one from line 3 to λ_3, one from line 4 to λ_1, and one from line 5 to λ_2. The event-based call graph of Figure 10 also contains several may-happen-before edges that are implied by the order in which the various expressions are executed. For instance, the `listen` on line 4 *may-happen-before* the `emit` on line 6, since the lines 4, 5 and 6 are executed in sequence.

4.2 Bug Finding

The event-based call graph enables us to discover a range of interesting program behaviors and potential bugs:

Dead Listeners: We can detect situations where an event listener is registered for an event τ on an object o, but the event τ is never emitted on o. This can happen for several reasons: (a) the event listener could be registered on the wrong object, (b) the event listener might be registered
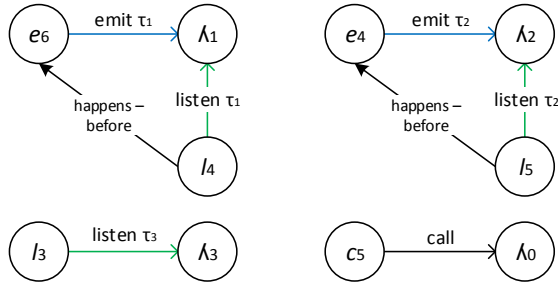
Figure 10. The event-based call graph for the λ_ϵ program in Figure 9. The call graph shows that λ_1 and λ_2 are executed due to the events τ_1 and τ_2, respectively. Furthermore, λ_0 is executed due to a regular function call, and λ_3 is never executed since the event τ_3 is never emitted.

for the wrong event, (c) the event might be emitted *before* the event listener is registered, and never emitted again.

Dead Emits: We can detect situations where an event τ is emitted on an object, but where that object has no event listener for τ. Such an event might be emitted by the application code or by a framework such as Node.js. If the event is emitted by the application, then the absence of a listener would seem unintentional and potentially indicates the presence of a bug. If, on the other hand, the event is emitted by the framework, then whether the event must be handled depends on the semantics of the event. In Node.js, some events are informational and can safely be ignored. However, an event such as the `connection` event on TCP sockets should *never* be ignored, if the socket is to accept connections.

Mismatched Synchronous/Asynchronous Calls: Most I/O operations are asynchronous in Node.js, but for some operations synchronous variants are also provided. For instance, reading a file can be done asynchronously with `readFile` and synchronously with `readFileSync`. If a file is accessed both synchronously and asynchronously there is likely a race condition. We can detect such races by inspecting the may-happen-before relation together with auxiliary information about the shared resource (e.g., the name of the file being accessed).

Unreachable Functions: We can detect functions that are unreachable, i.e., never executed directly or indirectly by the event system. This can help the programmer remove unused functions from the source code, or to detect functions which were actually supposed to be used, but forgotten at some point.

We observe that many of these properties require flow-sensitivity to determine the presence (or rather the absence) of a bug. For instance, an event listener is dead if the `listen` statement happens after the `emit` statement.

Example. Figure 11 shows a buggy λ_ϵ program. Figure 12 shows the event-based call graph for the program of Figure 11. The graph reveals several bugs in the program: First, λ_1 has an incoming listen edge, but no incoming emit edge, thus it is a *dead listener*. Second, the emit expression e_4 has no outgoing emit edge(s), thus it is a *dead emit*. Third, λ_2 has both incoming listen and emit edges, but the emit e_3 happens before the listen l_5, thus e_3 is a *dead emit* and l_5 is *dead listener*.

```
1  let (x = {})
2      x.listen(τ₁, λ₁ () ...);
3      x.emit(τ₂);
4      x.emit(τ₃);
5      x.listen(τ₂, λ₂ () ...)
```

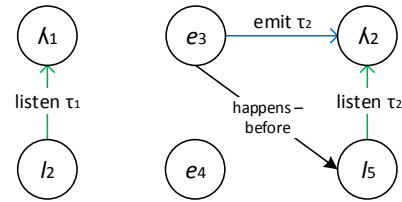**Figure 11.** A buggy λ_ϵ program.

Figure 12. An event-based call graph for Figure 11.

5. Analysis Framework

This section presents three analyses for computing event-based call graphs. These analyses were designed after studying a number of examples from the Node.js documentation and a few examples from StackOverflow. In Section 6 we evaluate these analyses on additional subject programs and report on their relative cost and precision.

Static Analysis. The exact call graph of a given program is uncomputable in general. We evaluate a static analysis framework for λ_ϵ that soundly over-approximates the exact event-based call graph. If a call, listener registration, or event emission occurs in some concrete execution, then the call graph must include it. However, the call graph may be conservative and contain *spurious* behaviour that does not actually occur in any execution. An analysis is said to be more precise than another if its output contains less spurious behaviour. In the case of analyses that construct event-based call graphs, this means that an analysis is more precise than another if it produces a call graph with fewer edges. For example, two emit edges $\mathtt{emit}_1 \hookrightarrow \lambda$ and $\mathtt{emit}_2 \hookrightarrow \lambda$ in a generated call graph indicate that either $\mathtt{emit}_1$ or $\mathtt{emit}_2$ may cause λ to be scheduled. However, a call graph that contains only $\mathtt{emit}_1 \hookrightarrow \lambda$ guarantees that only $\mathtt{emit}_1$ causes λ to be scheduled.

$$
\begin{aligned}
\widehat{\sigma} \in \widehat{Heap} &= \widehat{Addr} \to \widehat{Val} \\
\widehat{\vartheta} \in \widehat{Listener} &= \widehat{Addr} \times Event \to \mathcal{P}(Lam) \\
\widehat{\pi} \in \widehat{Queue} &= \widehat{Addr} \times Event \to \mathcal{P}(Lam \times \widehat{Val}) \\
\widehat{s} \in \widehat{State} &= \widehat{Heap} \times \widehat{Listener} \times \widehat{Queue} \times Exp \\
\ell \in Lattice &= Ctx \times Loc \to \widehat{State} \\
c \in Ctx &= \text{is a set of contexts.} \\
l \in Loc &= \text{is a set of source code locations.}
\end{aligned}
$$

Figure 13. Abstract runtime of λ_ϵ.

Design Motivation. Based on our experience with Node.js programs, we make three general observations that guide the design of the analysis framework:

Observation 1: It is common for one event listener to register another. A very common pattern is to perform some asynchronous I/O operation first. Then, when that operation completes, another I/O operation is performed, and so on, until the entire computation is complete.

Observation 2: It is common for event names to overlap between different objects. For example, in Node.js, the event names `connection`, `data` and `finished` occur in many contexts.

Observation 3: It is uncommon for a single object to have multiple event listeners registered for the same event.

Abstract Semantics. The static analysis framework is based on a parameterized abstraction of the concrete semantics of λ_ϵ presented in Figure 7. The abstract semantics is shown in Figure 13. The abstractions of runtime addresses *Addr* and primitive values *Val* are parameters that can be instantiated with different abstract domains. Our implementation uses the allocation site abstraction to abstract memory addresses and the constant propagation lattice to abstract values. Further details are provided in Section 6.

The most interesting aspect of the design space is how to abstract the map of registered listeners and the event queue. Recall that the map of registered listeners maintains, for every object and event, the order in which the listeners were added. Similarly, the event queue map maintains, for every object and event, which listeners are scheduled for execution. We choose a pragmatic abstraction that replaces both sequences by sets, forgetting the order in which event listeners were registered, or listeners should be scheduled *for the same event and on the same object*. However, we do maintain separation between different event listeners (and their associated events) and their execution.

Discussion. We sketch a few alternative abstractions and discuss why we ultimately decided on the current one. A more precise alternative would maintain the two sequences up to some bound, e.g., the order of the first k event listeners to be scheduled or executed. Such an abstraction would be more precise, since it can distinguish between the order in which event listeners are executed for the same object and event. On the other hand, it is less common in practice to have multiple listeners registered for the same event and object. A less precise abstraction could ignore on what specific object or event an event listener is registered. However, this would immediately lead to imprecision since different objects commonly share the same event names in Node.js.

Context-Sensitivity Policies. In a Node.js program, after the execution of initialization code, the runtime enters an event loop that repeatedly identifies events that have been emitted, schedules and executes corresponding listeners. To understand the temporal behavior of a Node.js program, it is important to understand how the event loop behaves in different contexts. For example, suppose that in a given program, listener A registers listener B for an event. A simplistic understanding would be that the event loop can call both listeners at any time. To identify that A must execute first, the analysis must consider the event loop first in the context before B has been registered, and separately in the context after B has been registered. To evaluate the precision of different context abstractions, the analysis framework is parameterized by a context-sensitivity policy. Context selection is driven by emitted events and registered event listeners. We have evaluated the following three policies:

1. Baseline Analysis. The *baseline analysis* ignores any dependencies between registered event listeners and emitted events. It collapses the event loop into a single program point where all dataflow is merged. In effect, this analysis assumes that listeners can run at any time, in an arbitrary order, and any number of times. The baseline analysis has only a single context $Ctx = \circ$. This analysis approach is inexpensive, but the precision benefits of flow sensitivity are lost. The baseline analysis corresponds to what is currently used in state-of-the-art static analyzers such as TAJS [13] and JSAI [18].

2. Event-Sensitive Analysis. The *event-sensitive* analysis separates dataflow based on which listeners have executed so far due to emitted events. This analysis captures the order in which event listeners, for different objects/events, are extracted from the (abstract) event queue $\widehat{\pi}$ and executed. In the event-sensitive analysis, a context consists of a set of object and event pairs:

$$ c = \{(o, \tau) \mid \tau \text{ has been emitted and executed on } o\} $$

For example, the context $\{(\tau_1, o_1), (\tau_1, o_2), (\tau_2, o_2)\}$ indicates that event τ_1 was emitted (and its event listeners executed) on objects o_1 and o_2, that τ_2 was emitted on o_2, and that no other events were emitted. The *event sensitive* context is given by the power set:

$$ Ctx = \mathcal{P}(\widehat{Addr} \times Event) $$

where the initial context is the empty set.

```
1  let (x = {})
2      x.listen(τ_1, λ_1 () x.emit(τ_2));
3      x.listen(τ_2, λ_2 () ...);
4      x.emit(τ_1)
```

Figure 14. A small λ_ϵ program which illustrates the increased precision due to *event sensitivity*.

Example. Figure 14 shows a small λ_ϵ program that illustrates the increased precision due to *event sensitivity*. Here, two event listeners λ_1 and λ_2 are registered for events τ_1 and τ_2 on some object. The evaluation of λ_1 emits an event that causes λ_2 to be scheduled. Thus, all side-effects of λ_1 precede λ_2. In the baseline analysis, this correlation is lost but the event sensitive analysis preserves the correlation by returning to the event loop in the context: $\{(\tau_1, o_1)\}$.

Remark: A critical reader may observe that the event-sensitive algorithm has a worst case exponential running time, and be concerned about its performance. As we shall see in Section 6, such exponential behavior does appear to arise in practice. The reason for proposing the event-sensitive analysis here is that it can be seen as the "obvious" solution" for capturing dataflow dependencies between event listeners.

3. Listener-Sensitive Analysis. The *listener-sensitive* analysis separates dataflow based on which listeners have been registered. In the listener-sensitive analysis, a context is a set of triples of an object o, an event τ, and a listener λ that is registered on that object and for that event:

$$c = \{(o, \tau, \lambda) \mid \lambda \text{ is registered on object } o \text{ for event } \tau\}$$

The *listener sensitive* context is given by the power set:

$$Ctx = \mathcal{P}(\widehat{Addr} \times Event \times Lam)$$

where the initial context is the empty set.

The strength of the listener-sensitive analysis is its ability to track listeners that register other listeners as part of their execution. But in general the precision of the event- and listener-sensitive analyses is incomparable. As will be shown later, the listener-sensitive policy avoids many spurious warnings for dead emits and dead listeners since it precisely separates dataflow based on what listeners are registered.

Example. Figure 15 shows an example demonstrating the benefits of listener-sensitive analysis. Here, two listeners for the same event τ, are registered, one following the other. The execution of λ_1 causes λ_2 to be registered for the same event, and on the same object, as λ_1. The baseline and event-sensitive analyses cannot capture the fact that λ_1's execution is known to precede λ_2's. This correlation, however, is captured by listener sensitivity: The event loop is unfolded into the two contexts: $\{(o_1, \tau, \lambda_1)\}$ and $\{(o_1, \tau, \lambda_1), (o_1, \tau, \lambda_2)\}$

```
1  let (x = {})
2      x.listen(τ, λ_1 ()
3          x.listen(τ, λ_2 () ...);
4          x.emit(τ)
5      );
6      x.emit(τ)
```

Figure 15. A small λ_ϵ program which illustrates the increased precision due to *listener sensitivity*.

Soundness. The baseline, event- and listener-sensitive analyses are all sound in the following sense: given a sound analysis of λ_ϵ, i.e., a sound analysis for λ_{JS} extended with the event constructs defined in this paper, the three context sensitivity policies are special cases of the trace partitioning framework of Rival and Mauborgne [24]. Intuitively, separation of data flow in the event loop cannot cause the analysis to miss any error, since if there is an error, it must occur in at least one of the contexts. Furthermore, if the separation, i.e., the trace partition, is finite, then the analysis is guaranteed to terminate.

6. Evaluation

In this section, we discuss a tool that implements the analyses described in Section 5, and report on experiments in which the tool is applied to small Node.js programs.

6.1 Implementation

We have implemented a static analysis tool, RADAR, for Node.js applications written in JavaScript. RADAR can detect errors such as the ones discussed in Section 2 and supports most of the JavaScript language, including higher-order functions, prototype inheritance and dynamic property access. Similar to other work on static analysis of JavaScript, RADAR does not support the `with` and `eval` constructs.

The implementation consists of approximately 27,000 lines of Scala code, of which around 2,000 lines are related to the event system. The implementation has been used in previous work on dynamic field access [21] and sparse dataflow analysis [22]. All experiments were performed on an Intel Core i5-4300U 2.5GHz CPU with 2GB memory allocated for the JVM.

Analysis Details. In essence, our analysis is a flow-sensitive dataflow analysis. Flow-sensitivity means that the order of statements is respected, which is important for determining *may-happen-before* relationships between, e.g., `emit` and `listen` statements. The analysis incorporates a field-sensitive subset-based points-to analysis. Field-sensitivity means that the points-to analysis distinguishes between the values of fields with different names. The points-to analysis models the (potentially) unbounded heap using the allocation site abstraction, meaning that objects allocated at the same location in the source code are represented using the same abstract summary object. The analysis constructs the event-based call graph on-the-fly due to both indirect

Module	Lines	Functions	listen	emit
Filesystem	508	73	31	31
Http	321	45	24	13
Network	328	39	31	36
Stream	111	12	2	19
Total	1,268	169	86	96

Table 1. The Node.js model. The four columns show, for each module, the number of lines of code, functions, and listen and emit statements in the stub model.

calls and events. Specifically, the base object(s) of `emit` and `listen` statements are not known ahead-of-time, but gradually discovered as the points-to graph is resolved.

Node.js Model. We model the Node.js framework with JavaScript stubs that use the `listen` and `emit` constructs. This has three consequences: First, it demonstrates that our framework is sufficiently powerful to capture the semantics of Node.js. Second, it makes the analysis simpler to implement since events in the application and the library can be treated uniformly. Third, it makes it easy to model other event-based libraries such as `socket.io` and `async.js`.

The Node.js core consists of around thirty modules. Of these, we have modeled `filesystem`, `http`, `network` and `stream`, which make the most heavy use of events; Modeling other modules is straightforward. We stress that only core modules (and other native modules implemented in C++) require modeling. Specifically, if a Node.js module is implemented in pure JavaScript it can be directly fed to the analysis. Table 1 shows some statistics about the models. In total, the four modules define 169 functions containing 86 listen and 96 emit statements.

6.2 Research Questions

We evaluate RADAR based on two research questions:

Q1: Is the tool useful for finding and understanding event-related bugs?

Q2: What is the precision and performance of the tool?

Sections 6.3 and 6.4 address these research questions in detail, for each of the *baseline*, *event-sensitive* and *listener-sensitive* analyses.

6.3 Q1: Finding and Understanding Bugs

In order to determine whether RADAR can help the programmer find event-related bugs, we apply it to buggy program fragments obtained from StackOverflow, a popular question-and-answer forum for software developers. In each case, the programmer supplied a program fragment and a description of the intended behavior. We selected twelve StackOverflow questions based on the criterion that the program had to contain a single bug, and that this bug was related to the way events are emitted and listeners are registered. We used all

	BASELINE		EVENT		LISTENER	
Question	Orig.	Fixed	Orig.	Fixed	Orig.	Fixed
11790224	✓	✗	✓	✓	✓	✓
13338350	✓	✗	✓	✓	✓	✓
16903844	✓	✓	✓	✓	✓	✓
17894000	✓	✓	✓	✓	✓	✓
18295923	✓	✓	✓	✓	✓	✓
19081270	✓	✗	✓	✓	✓	✓
19167407	✓	✓	✓	✓	✓	✓
19171045	✓	✓	✓	✓	✓	✓
19342910	✓	✓	✓	✓	✓	✓
23437008	✓	✓	✓	✓	✓	✓
25650189	✓	✗	✓	✓	✓	✓
26061335	✓	✗	✓	✓	✓	✓

Table 2. Twelve buggy program fragments from the Stack-Overflow. A ✓ in the `Orig.` column means that the analysis correctly reported the bug for the original program. A ✗ in the `Fixed` column means that the analysis still reported a (spurious) warning for the fixed (corrected) program. All analyses completed in less than a second.

StackOverflow code fragments that we could find that fit this criterion. We evaluate each analysis by applying it to the program and observing:

1. Does the analysis report a warning? How useful is the warning for finding the bug?

2. Does the warning disappear once the program has been corrected? If the warning remains, due to analysis imprecision, then that may confuse the programmer.

Table 2 shows the results for each analysis. The most interesting program fragments are discussed further below:

• In Q13338350, entitled "NodeJS writeStream empty file", the programmer wants to write the content of a buffer to a file stream. The programmer opens the file stream, calls `write` passing the buffer and calls `end` to close the file stream. However, the call to `end` immediately closes the stream and does not wait for the `write` operation to complete. As a result, an empty file is produced. The fixed program listens for the `end` event to be emitted on the stream (signifying that there is no more data to be *read* from the buffer) and then calls `close` on the stream. All three analyses report that the `close` operation possibly happens before the `write` operation. On the fixed program, the baseline analysis still spuriously reports that the close operation may happen before the write operation completes. The event- and listener sensitive analyses report no spurious warning.

• In Q17894000, entitled "event handling not working as expected nodejs", the programmer creates a socket, registers a listener for the `data` event and inside that listener registers another listener for the `close` event. The programmer then expects the listener for the `close` event to be executed. However, it is permissible for a socket to be closed before

Program	Lines	Nodes	BASELINE			EVENT-SENSITIVE			LISTENER-SENSITIVE		
			D. Listener	Mhb	Time	D. Listener	Mhb	Time	D. Listener	Mhb	Time
Filesystem	60	186	1	3	0.8s	0	2	0.9s	0	2	0.9s
Http	250	618	10	66	2.5s	–	–	–	0	41	3.0s
Network #1	200	546	8	61	2.2s	–	–	–	1	38	3.7s
Network #2	170	463	6	54	1.7s	0	33	22.1s	0	33	1.6s
Node in Action, ch. 3	330	668	8	49	2.5s	–	–	–	1	27	6.0s
Node in Action, ch. 4	390	873	10	81	3.5s	–	–	–	3	66	17.0s

Table 3. Analysis results for six programs; four from the Node.js documentation [5] and two from the book *Node.js in Action* [4]. The `Lines` and `Nodes` columns show the number of source code lines and nodes in the control-flow graph. The `D. Listener`, `Mhb` and `Time` columns show, for each analysis, the number of dead listeners reported, the number of may-happen-before relations and the total analysis time. The double dashed line, –, indicates that the analysis timed out after 60 seconds.

any data is transmitted, and thus there is no guarantee that the `close` listener is ever registered. Each of the three analyses identifies the problem by warning that `close` may be emitted before the listener is registered. Moving the listener registration for the `close` event out of the listener for the `data` event fixes the problem, and none of the three analyses report any spurious warnings after the fix.

• In Q18295923, entitled "nodejs stdin readable event not triggered", the programmer registers a listener for the `readable` event on the standard-in `process.stdin` stream. The programmer then expects the listener to be executed when the user writes to standard-in. However, the `process.stdin` stream is by default in a *paused* mode where the programmer must call `resume()` before the `readable` events are emitted. All three analyses report a dead listener, and no spurious warning on the fixed program.

• In Q19081270, entitled "Why my fs.readFileSync does not work", the program writes a file asynchronously, but then immediately reads back the file synchronously, before the asynchronous operation has had a chance to complete. (Node.js provides both asynchronous and synchronous variants of many operations, and it is easy for programmers to get confused and make mistakes such as this one). The analyses do not have any knowledge of the external world (e.g., the file system), so they cannot directly pinpoint the error. They can, however, report that the read always occurs before the write, which can help the programmer understand the problem. The recommended fix is to register an asynchronous listener that executes when the write completes. This solves the problem, but the baseline analysis still reports that the read may execute before the write, since it cannot accurately track the may-happen-before dependencies.

• In Q19167407, with the title "Data event not firing NodeJS", the programmer creates a readable stream and a writable stream, and pipes data from the readable stream to the writable stream. The programmer registers listeners for the `data` and `close` events on the writable stream. However, the listener for the `data` event is never executed. The problem is that only the readable stream emits `data` events, not

the writable stream: The programmer has registered the listener on the wrong object. All three analyses report a dead listener and report no spurious warnings when applied to the fixed program.

• In Q19342910, entitled "when is the `connect` event in nodejs net module emitted?", the programmer creates a TCP server object passing in a listener function. Inside that function the programmer registers a listener for the `connect` event. However, the inside listener is never executed. The problem is that function, which is passed during the construction of the TCP server object, is implicitly registered for the `connect` event. Thus, once the `connect` event is emitted, the outer function executes, which then registers a listener for the very same event. Since the `connect` event is only emitted once (per connection), the inner listener is never executed. All three analyses report the inner listener as dead and no spurious warnings on the fixed program.

We conclude that the analyses appear to be useful for finding event-related bugs in small Node.js programs. However, the baseline analysis reports many spurious warnings. The event- and listener-sensitive analyses each provide sufficient precision to avoid reporting any spurious warnings on these small programs.

6.4 Q2: Precision and Performance

To answer this question, we apply each analysis to 6 programs; 4 from the online Node.js API documentation and 2 from the book: *Node.js in Action*. The programs from the Node.js documentation illustrate the `filesystem`, `http` and `network` modules. The *Node.js in Action* programs show how to perform a HTTP request and create a small HTTP server. Although these programs are small[6], their sizes are consistent with benchmarks used in recent literature on flow-

---

[6] We selected small programs due to the scalability challenges faced by any flow-sensitive pointer analysis. Our implementation is based on [21, 22] in which progress was made, but scalability challenges remain. However, it should be noted that our event-based techniques do not depend on a particular analysis framework and should remain applicable in the future, when better JavaScript analysis frameworks become available.

sensitive, subset-based points-to analysis for JavaScript [13, 18, 22].

We report, for each analysis, the number of warnings for dead listeners, the number of may-happen-before relations and the analysis time. The results, shown in Table 3, expose significant differences in the scalability and precision of the three analyses.

As an example, the "Filesystem" example is 60 lines of code (for the program and relevant parts of the Node.js model). The baseline analysis spuriously reports one dead listener and three may-happen-before edges in 0.8 seconds, both the event-sensitive and listener-sensitive analyses report no dead listeners and two may-happen-before edges in 0.9 seconds. Thus, these two analyses avoid the spurious warning reported by the baseline analysis.

The event-sensitive analysis times out on 4 benchmarks and runs slowly on another. An explanation for the poor performance, compared to the listener-sensitive analysis, is that it unnecessarily separates abstract states. E.g., if the event queue contains $\{\tau_1, \tau_2\}$, the event-sensitive analysis may separate dataflow for the states $\{\tau_1\}$, $\{\tau_2\}$ and $\{\tau_1, \tau_2\}$, even if the same listeners are present in each case. This can lead to an exponential blow-up in the number of states. The event-sensitive analysis, although conceptually simple, is too inefficient in practice.

The listener-sensitive analysis offers a compelling alternative. Compared to the baseline analysis, the number of may-happen-before edges decreases from 314 to 207, showing that flow-sensitive precision is increased significantly, resulting in a reduction in the number of (spurious) warnings from 43 to 8. Analysis times increase by a factor from 1.0x to 5.0x. Thus, for a reasonable increase in analysis time, the number of spurious warnings is reduced significantly.

We examined the remaining spurious warnings reported by the listener-sensitive analysis, and found two causes:

- We model asynchronous callbacks by creating a fresh object, registering the callback on that object for some fixed event τ, and then emitting τ. In some cases this object became summarized and the analysis was unable to determine that the registration happened before the emit. We believe this problem could be overcome with heap sensitivity, i.e. by splitting allocation sites based on the current context.

- We chose to abstract the *listener ordering* for event listeners registered on the *same object* and for the *same event* with a set. Thus, if λ_1 is registered for some event τ and λ_2 is later registered for the same event, we lose the ordering between λ_1 and λ_2. In a single instance this caused a loss of precision.

In summary, the listener-sensitive analysis is preferable to the baseline- and event-sensitive analyses; the baseline analysis produces too many spurious warnings whereas the event-sensitive analysis has poor performance.

7. Discussion

Thus far, we have presented a technique for detecting errors related to event-handling and demonstrated its effectiveness for the specific case of the Node.js event model. While Node.js is a highly popular and rapidly growing platform, the reader may wonder about the applicability of our technique beyond this limited setting. In this section, we briefly consider the issues that arise other settings.

7.1 JavaScript in the Browser Environment

The original and dominant use of JavaScript remains web applications that run inside a web browser. Here, events are emitted in response to user actions (e.g., mouse clicks or keyboard presses) and internal browser events (e.g., the completion of network requests). Events are propagated along the structure of the HTML document using the *event capturing* and *event bubbling* propagation strategies[7] and listeners are attached to HTML elements and can cancel (i.e., stop propagation) of events programmatically.

We believe that the browser environment poses three distinct research challenges for any static analysis:

1. How to model the structure of the HTML document and its associated operations?

2. How to model the event propagation strategies: event bubbling and event capturing?

3. How to capture the data– and control-flow dependencies between event listener registrations and event emissions?

The first of these challenges, precise modeling of the DOM data structure that reflects the current HTML document, remains a significant hurdle in practice, although some progress has been made in [13]. We expect that the second challenge, modeling the event propagation strategies, can be handled by way of new `listen` and `emit` constructs possibly in combination with the model of [19]. The third challenge is exactly the same as in the case of Node.js, so we expect our technique to be directly applicable to traditional JavaScript web applications in combination with suitable solutions for the first two challenges, as these become available.

7.2 Other Languages

In this paper, we have defined the event-based call graph, which extends of the traditional notion of a call graph with nodes and edges that reflect the flow of control due to events. Furthermore, we have defined context-sensitive static analyses for computing approximations of the event-based call graph, which make use of context-sensitivity policies that separate data flow based on the set of events that have been emitted and the set of event listeners that have been registered. While the work in this paper has been focused on JavaScript, we believe that our ideas could in principle be

---

[7] http://www.quirksmode.org/js/events_order.html

applied to other dynamic languages such as Python [20] and Ruby [32] where event handling is commonly used. This would require modeling language-specific idioms for registering listeners and emitting events in terms of the `listen` and `emit` constructs of Section 3. It should be noted that our techniques could do not require that event-handling is provided as a first-class language construct and that they can be applied to implementations of event-handling that are provided as a library or API, as appears to be common practice in the case of both Python and Ruby.

8. Related Work

To our knowledge, little prior work exists on whole-program *static analysis* of *event-based* JavaScript code. We discuss two categories of related work: static analysis of JavaScript, and dynamic analysis of event-based JavaScript code.

Static Analysis of JavaScript. Guarnieri et al. present GATEKEEPER, a tool for enforcing security policies for JavaScript widgets [9]. GATEKEEPER used one of the first points-to analyses for JavaScript to discover dataflow disallowed by a security policy.

Jensen et al. present a dataflow analysis, TAJS, for browser-based JavaScript applications [13]. The focus is on how to represent the DOM, but events are discussed briefly. Their analysis conservatively assumes that any browser event, e.g., `onclick`, may execute at any time and makes no attempt, other than for *onload*, to separate dataflow based on what listeners are registered. By contrast, our baseline analysis is more precise, as it only considers events that are actually emitted. More recently, TAJS was extended to handle certain cases of `eval` [14] and to analyze the popular jQuery library [1].

Zheng et al. present a static analysis for detecting race conditions related to asynchronous AJAX requests [34]. Specifically, a race condition exists when an event listener may read a global variable, at any time, and an asynchronous response listener may write to the same global variable. In relation to our work, Zheng et al. are not concerned about the *specific* order in which event listeners are executed, but the existence of any order that could potentially cause a race.

Dynamic Analysis of Event-Based JavaScript Programs. There have been several threads of research in which dynamic analysis is employed to find data races and asynchrony-related errors in JavaScript programs.

Lerner et al. present a model of the event mechanism in the DOM which incorporates the *event bubbling* and *event capturing* propagation strategies together with event cancellation [19]. Based on the model, the authors automatically construct test cases which are used to detect implementation inconsistencies across different browsers. A key difference between between their model and ours is determinism. Our model, which is motivated by asynchronous events in Node.js, must be inherently non-deterministic. On the other

hand, the model of Lerner et al. is deterministic since event dispatch (capturing/bubbling) is deterministic. An interesting venue for future work would be a combination of the two models, but this is beyond the scope of the present work, and not required for analysis of Node.js applications.

Artzi et al. present a framework for feedback-directed random testing of JavaScript applications [2]. This framework keeps track of event-handler registrations and attempts to increase code coverage by generating sequences of events to execute unexplored code. Artzi's work aimed to find execution errors and situations where malformed HTML was created, and did not consider event-handling related errors.

Petrov et al. define a *happens-before* relationship on the various kinds of operations performed by web applications (HTML parsing, access to variables and DOM nodes, event-handler execution) [28]. A notion of logical memory locations is defined, to abstract both JavaScript heap locations and locations in browser-specific native data structures. A *data race* is defined as a situation where two operations access the same logical memory locations, at least one of these accesses is a write, and no ordering exists between the operations. Petrov et al. implemented WebRacer, a dynamic detector for such races, and used WebRacer to find races in sites of Fortune 500 companies. Petrov's happens-before relation includes orderings between operations corresponding to `emit` and `listen` edges, and the event-dispatch races detected by WebRacer resemble our StackOverflow examples.

Raychev et al. observe that race detection techniques such as the one of Petrov et al. report an overwhelming number of races in cases where event handler execution is coordinated via shared variables [29]. They reduce the number of false positives by introducing a notion of *race coverage* where a race a covers a race b if treating a as synchronization eliminates b as a race. Raychev et al. implemented their work in a tool called EventRacer, and showed that, by focusing the user's attention on uncovered races, the number of issues that need to be considered is reduced dramatically.

The position paper by Mutlu et al. is focused on *observable races* in web applications that have visually apparent symptoms (e.g., missing elements in a user-interface) [27]. To this end, Mutlu et al. employ various strategies to instrument an application so that delays are inserted at each XML-HttpRequest. By comparing the screenshots generated in each case, observable races are detected. Mutlu et al. propose different strategies for inserting delays—randomly, or systematically in a way that allows all schedule permutations to be exercised.

Hong et al. present WAVE, a framework for detecting various concurrency-related errors in client-side JavaScript code [11]. Here, an execution is monitored to create an execution model. From this model, test cases are generated that permute the order of operations in the original execution. A problem is reported if a test raises an exception, does not terminate, or produces a result different from that of the origi-

nal execution. Among the types of problems that WAVE detects are data races like the ones found by EventRacer [29], and atomicity violations.

Static Analysis of Asynchronous Programs. Jhala et al. present an interprocedural dataflow analysis framework for asynchronous programs [15]. The framework is based on an extension of the IFDS framework by Reps et al. [30]. A key feature of this framework is that calls and returns are well-matched with the requirement that the transfer functions are distributive. In contrast, the monotone framework [16], which our analysis is based on, imposes no such requirements, but at the same time may have mismatched call/returns. As argued by Jensen et al. and Kashyap et al. the challenges posed by JavaScript require sophisticated lattices which are rarely distributive and consequently not expressible in IFDS [12, 18]. For a concrete example, IFDS assumes that the call graph is known, whereas JavaScript analyses such as TAJS, JSAI and the present work construct the call graph on-the-fly.

9. Conclusion

We have introduced the *event-based call graph* and shown that it is useful for finding various event-related bugs in Node.js applications. The event-based call graph incorporates information about listener registration and event emission and can be used to detect *dead listeners* or *dead emits*. We have designed and implemented three sound analyses, a *baseline*, an *event sensitive* and a *listener sensitive*, to compute the event-based call graph. The baseline analysis is equivalent to what has been used in previous work on static analysis of JavaScript applications. Experimental results show that the analyses are able to find bugs in small buggy programs posted on the StackOverflow website. Furthermore, the experiments suggest that the baseline analysis, used in prior work, produces many false positives. On the other hand, the listener sensitive analysis, which separates dataflow based on the set of currently registered listeners, offers significantly better precision at a modest increase in analysis time.

References

[1] E. Andreasen and A. Møller. Determinacy in Static Analysis for jQuery. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2014.

[2] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A Framework for Automated Testing of JavaScript Web Applications. In *Proc. 33rd International Conference on Software Engineering (ICSE)*, 2011.

[3] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Fault Localization for Dynamic Web Applications. In *IEEE Transactions on Software Engineering*, 2012.

[4] M. Cantelon, M. Harter, T. Holowaychuk, and N. Rajlich. *Node.js in Action*. Manning Publications, 2014.

[5] R. Dahl. Node.js online documentation, 2014.

[6] A. Feldthaus and A. Møller. Semi-Automatic Rename Refactoring for JavaScript. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

[7] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported Refactoring for JavaScript. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[8] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.

[9] S. Guarnieri and B. Livshits. GateKeeper: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *Proceedings of the Usenix Security Symposium*, 2009.

[10] A. Guha, C. Saftoiu, and S. Krishnamurthi. The Essence of JavaScript. In *Proc. 24th European Conference on Object-oriented Programming (ECOOP)*, 2010.

[11] S. Hong, Y. Park, and M. Kim. Detecting Concurrency Errors in Client-Side JavaScript Web Applications. In *Proc. of 17th International Conference on Software Testing, Verification and Validation (ICST)*, 2014.

[12] S. H. Jensen, A. Møller, and P. Thiemann. Type Analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS)*, 2009.

[13] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In *Proc. 8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2011.

[14] S. H. Jensen, P. Jonsson, and A. Møller. Remedying the Eval That Men Do. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[15] R. Jhala and R. Majumdar. Interprocedural Analysis of Asynchronous Programs. In *Proc. 34th Symposium on Principles of Programming Languages (POPL)*, 2007.

[16] J. B. Kam and J. D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Informatica*, 1977.

[17] V. Kashyap, J. Sarracino, J. Wagner, B. Wiedermann, and B. Hardekopf. Type Refinement for Static Analysis of JavaScript. In *Proc. 9th Symposium on Dynamic Languages*, 2013.

[18] V. Kashyap, K. Dewey, E. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. JSAI: A Static Analysis Platform for JavaScript. In *Proc. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[19] B. S. Lerner, M. J. Carroll, D. P. Kimmel, H. Q.-D. L. Vallee, and S. Krishnamurthi. Modeling and reasoning about dom events. In *Proc. 3rd USENIX Conference on Web Application Development*, 2012.

[20] M. Lutz. *Learning Python*. O'Reilly, 5 edition, 2013.

[21] M. Madsen and E. Andreasen. String Analysis for Dynamic Field Access. In *Proc. 23rd International Conference on Compiler Construction (CC)*, 2014.

[22] M. Madsen and A. Møller. Sparse Dataflow Analysis with Pointers and Reachability. In *Proc. 21st International Static Analysis Symposium (SAS)*, 2014.

[23] M. Madsen, B. Livshits, and M. Fanning. Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries. In *Proc. European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2013.

[24] L. Mauborgne and X. Rival. Trace Partitioning in Abstract Interpretation based Static Analyzers. In *Proc. 14th European Symposium on Programming (ESOP)*, 2005.

[25] F. Meawad, G. Richards, F. Morandat, and J. Vitek. Eval Begone!: Semi-automated Removal of Eval from JavaScript Programs. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2012.

[26] A. Møller and M. Schwarz. Automated Detection of Client-State Manipulation Vulnerabilities. *Transactions on Software Engineering and Methodology*, 2014.

[27] E. Mutlu, S. Tasiran, and B. Livshits. I Know It When I See It: Observable Races in JavaScript Applications. In *8th Workshop on Dynamic Languages and Applications.*, 2014.

[28] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby. Race Detection for Web Applications. In *Proc. 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[29] V. Raychev, M. Vechev, and M. Sridharan. Effective Race Detection for Event-driven Programs. In *Proc. of 27th European Conference on Object-oriented Programming (ECOOP)*, 2013.

[30] T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proc. 22nd Symposium on Principles of Programming Languages (POPL)*, 1995.

[31] M. Schaefer, M. Sridharan, J. Dolby, and F. Tip. Effective Smart Completion for JavaScript. Technical Report RC25359, IBM Research, 2013.

[32] D. Thomas, A. Hunt, and C. Fowler. *Programming Ruby 1.9 & 2.0: The Pragmatic Programmer's Guide*. Pragmatic Bookshelf, 4 edition, 2013.

[33] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and Scalable Security Analysis of Web Applications. In *Proc. 16th International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2013.

[34] Y. Zheng, T. Bao, and X. Zhang. Statically Locating Web Application Bugs Caused by Asynchronous Calls. In *Proc. 20th International Conference on World Wide Web*, 2011.