Learning How to Listen: Automatically Finding Bug Patterns in Event-Driven JavaScript APIs

ELLEN ARTECA, Northeastern University, USA
MAX SCHÄFER, GitHub, UK
FRANK TIP, Northeastern University, USA

Event-driven programming is widely practiced in the JavaScript community, both on the client side to handle UI events and AJAX requests, and on the server side to accommodate long-running operations such as file or network I/O. Many popular event-based APIs allow event names to be specified as free-form strings without any validation, potentially leading to *lost events* for which no listener has been registered and *dead listeners* for events that are never emitted. In previous work, Madsen et al. presented a precise static analysis for detecting such problems, but their analysis does not scale because it may require a number of contexts that is exponential in the size of the program. Concentrating on the problem of detecting dead listeners, we present an approach to *learn* how to correctly use event-based APIs by first mining a large corpus of JavaScript code using a simple static analysis to identify code snippets that register an event listener, and then applying statistical modeling to identify unusual patterns, which often indicate incorrect API usage. From a large-scale evaluation on 127,531 open-source JavaScript code bases, our technique was able to detect 75 incorrect listener-registration patterns, while maintaining a precision of 90.9% and recall of 7.5% over our validation set, demonstrating that a learning-based approach to detecting event-handling bugs is feasible. In an additional experiment, we investigated instances of these patterns in 25 open-source projects, and reported 30 issues to the project maintainers, of which 7 have been confirmed as bugs.

Additional Key Words and Phrases: static analysis, JavaScript, event-driven programming, bug finding, API modeling

ACM Reference Format:
Ellen Arteca, Max Schäfer, and Frank Tip. 2021. Learning How to Listen: Automatically Finding Bug Patterns in Event-Driven JavaScript APIs. 1, 1 (March 2021), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Event-driven programming has been the dominant paradigm in JavaScript since its early days. This is quite natural on the client side, since most web applications are GUI-based and hence are centered around reacting to user actions such as clicking a button or pressing a key. The W3C UI Events standard [12] defines the low-level event API supported by all modern browsers, while popular libraries such as jQuery [19], Angular [7] and React [17] provide higher-level abstractions on top of it. Many other client-side APIs such as Web Workers and Web Sockets are likewise programmed in an event-driven style. On the desktop, the popular Electron [24] framework enforces an architecture where applications are split into a main process and a renderer process, which communicate via an event-based API. Finally, the Node.js

Authors' addresses: Ellen Arteca, arteca.e@northeastern.edu, Northeastern University, Boston, USA; Max Schäfer, max-schaefer@github.com, GitHub, Oxford, UK; Frank Tip, f.tip@northeastern.edu, Northeastern University, Boston, USA.

platform [20], which is dominant in server-side JavaScript, advocates an asynchronous programming style centered around a collection of event-based APIs for accessing resources like the file system, the network, or databases.

The precise APIs implemented by individual platforms and frameworks differ, but a common feature across all of JavaScript is the notion of a central *event loop* that handles event dispatching. Events are identified by an *event name* and may optionally have a payload. When an event happens, it is associated with a particular object, which is known as the *event target* in many client-side frameworks and the *event emitter* in Node.js. We will follow the latter terminology in this paper. Client code can register *listener functions* (or *listeners* for short) for a particular event on an event emitter. When an event is emitted, all the listener callbacks registered for it on the emitter object are run in sequence. While many events are emitted by framework code, application code can also emit events explicitly.

Most of the event-based APIs mentioned above are intrinsically *dynamic* and *untyped*. By "dynamic" we mean that the association between events and listeners can change over time, with new listeners being registered and existing listeners being removed throughout an event emitter's lifecycle. Indeed, it is common for listeners themselves to register or remove listeners on their own or on other emitter(s). By "untyped" we mean that event names are free-form strings that are not validated in any way, and can be associated with any emitter and any payload. In particular, client applications can emit and listen for custom events on emitters defined by a library.

While these two properties are prized by some for their flexibility, they also give rise to several classes of subtle bugs [30]. For example, if a listener registration misspells the name of the event or registers the listener on the wrong object, the listener will never be invoked. This is known as a *dead listener*. Dead listeners can also arise if a listener is registered at the wrong time, for instance after the event has already been emitted. The dual of a dead listener is a *lost event*, which can happen if an event emission misspells the event name or emits it on the wrong object. Both dead listeners and lost events are particularly hard to debug, as they manifest in the lack of execution of the listener function rather than an explicit error message.

In this paper, we concentrate on dead-listener bugs. Our goal is to detect such bugs *automatically* and *statically*, that is, without having to run the code under analysis.

Prior work by Madsen et al. [30] employs context-sensitive static analysis techniques to infer a semantic model of event emission and listener registration to identify dead listeners. Unfortunately, their analysis does not scale well because it may require a number of contexts that is exponential in the size of the program.

We propose instead to *learn* how to use event-based APIs by first mining a large corpus of JavaScript code with a simple static analysis to identify code snippets that register an event listener, and then applying statistical modeling to identify unusual patterns. Intuitively, if we look at enough code we would expect most API usages to be correct, so particularly rare patterns are likely bugs. We formalize this concept of "particularly rare" as thresholds in our statistical model, and identify patterns that meet these thresholds as potential bugs. Using the same thresholds, our approach also addresses the dual problem of learning *correct* uses, as we would expect "particularly common" uses of the APIs to be correct.

Figure 1 visualizes our approach. The top of the figure shows how models of event-driven APIs are constructed in two steps: First, a *data mining* analysis is applied to a large number of JavaScript projects to obtain a list of event listener registrations. These are represented as *listener-registration pairs* $\langle a, e\rangle$, where a represents an event emitter object symbolically (via an *access path* [32], see Section 3), and e is the name of the event the listener is registered for. The second step is *classification*, i.e., building a statistical model of the occurrence distributions of e's and a's, and using this to identify pairs $\langle a, e\rangle$ where the access path a and event e are rare relative to each other. In other words, we look for cases where e is rarely listened for by a, and a rarely registers a listener for e.

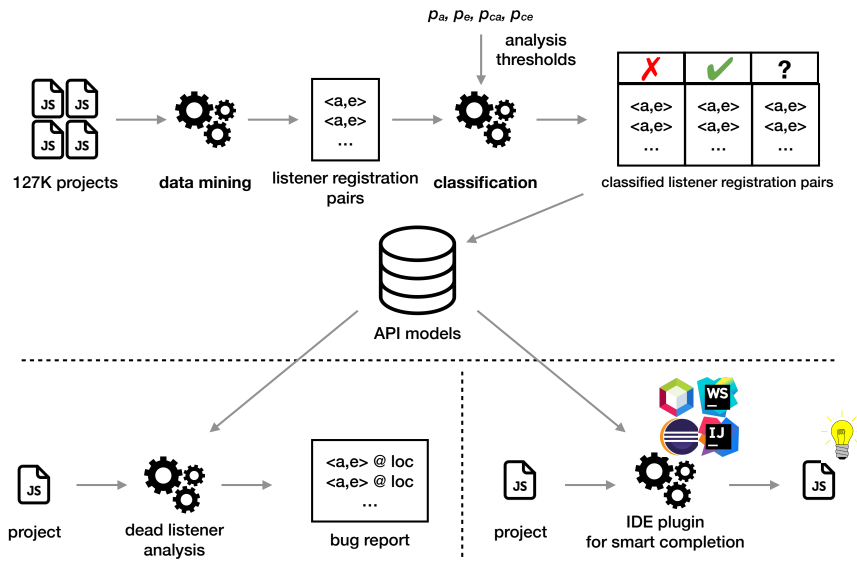

Fig. 1. Overview of approach.

Considering one of these conditions in isolation, or only the absolute number of occurrences of a pair, is not usually sufficient, since the data may be too sparse to conclude that it is incorrect. For example, a may represent a rarely-used API, or e may be a custom event that is only used by one particular code base. If, however, both the event emitter and the event name are rare for each other, then that is a strong indication that this pair represents a mistake.

Our statistical model has four parameters shown as inputs to the classification stage in Figure 1: rarity thresholds p_a and p_e defining when paths and events are considered rare, respectively, and confidence thresholds p_{ca} and p_{ce} defining the statistical confidence we demand for paths and events to be considered rare, respectively. The output of classification is a set of pairs learned to be correct, and a set learned to be incorrect. Pairs are left unclassified if they do not meet the thresholds for being common or rare.

These sets constitute API models, for those APIs analyzed. Once constructed, these API models can be used, e.g., in bug finding tools (see bottom left part of Figure 1), or for for smart completion in an IDE (see bottom right part of Figure 1). In this work, we focus on the set of pairs that are learned to be incorrect, as it is likely to indicate dead listener bugs.

The effectiveness of our approach crucially depends on how we configure the threshold parameters for categorization. In our evaluation, we systematically explore the space of possible configurations, computing for each of them the set of unusual listener-registration pairs from more than 532,000 pairs mined from over 127,500 open-source code bases. To quantitatively assess the quality of the models generated with a particular configuration, we then compute the true-positive rate (the *precision*) and the percentage of true positives detected (the *recall*) with respect to a smaller set of pairs that we semi-automatically labeled as either correct or incorrect.

In general, configurations with lower precision yield higher recall. For practically useful tools, however, a precision of at least 90% is generally considered essential [43, 44]. We show that several configurations achieve this rate over our labeled set.

To gain confidence that this is not simply an artifact of our data, we performed a 10-fold cross-validation experiment. We partitioned our labeled set into 10 sets; then for each of these sets we found the optimal configuration for the other 9 sets (which together form the training set), and computed the precision and recall of that configuration over the remaining (validation) set. Our results show that the optimal configuration for the training set consistently achieves good results over the validation set.

To qualitatively assess the usefulness of our approach, we used the constructed API models to look for dead listeners in 25 open-source projects, reporting 30 issues to the project maintainers. At the time of writing, 7 of these issues have been confirmed as bugs, and one has been patched.

The rest of the paper is structured as follows. Section 2 provides background on event-driven JavaScript programming and reviews a dead-listener bug in an open-source project. Sections 3 and 4 explain our approach in detail, while Section 5 covers the implementation. Section 6 presents our quantitative and qualitative evaluation, and discusses threats to validity. Section 7 reviews related work, and Section 8 concludes and outlines directions for future work.

Our code, data, and instructions on usage/reproducibility is included at: https://github.com/emarteca/JSEventAPIModelling

2 BACKGROUND

We begin by recapitulating the basics of event-driven programming in Node.js and some of the most common kinds of mistakes programmers make when writing event-driven code. We then show a concrete example of such a bug, based on code we found using our approach in an open-source project on GitHub, and finally explain how we go about identifying this sort of bug automatically.

2.1 Event-driven programming in Node.js

All event emitters in Node.js are instances of the `EventEmitter` class or one of its subclasses. Listeners are associated with an event by invoking one of several listener registration methods (such as `on` or `addListener`); these all take two arguments: an event name, which is a free-form string, and the listener function itself. Events can be emitted by invoking the `emit` method, which takes as its first argument an event name; any further arguments are passed as arguments to the listener functions associated with the event.

A typical example of this event-driven style is the `request` function from the `http` package in the Node.js standard library. Normally invoked as `http.request(url, fn)` where `url` is the URL to make a request to, and `fn` is a listener function, it creates an event emitter of type `http.ClientRequest` representing the pending request to `url` and associates `fn` with the `response` event of the request.

When a response to the request is received, the `response` event is emitted, causing `fn` to be invoked with an argument of type `http.IncomingMessage` representing the HTTP response. This object is itself an event emitter, emitting `data` events when response data becomes available and an `end` event once all data has been received.

If, on the other hand, the request times out before a response is received, the request object emits a `timeout` event.

2.2 Motivating example

Consider the code shown in Figure 2. It is based on a real-world bug our approach identified in the `min-req-promise` npm package, with all inessential details stripped away.

```
1  const http = require('http');
2  module.exports.request = (url) =>
3    new Promise((resolve, reject) => {
4      const req = http.request(url, res => {
5        res.on('data', /* omitted */);
6        res.on('end', () => {
7                  /* omitted */
8                  resolve( res);
9        });
10       res.on('timeout', () => reject(req)); // bug here
11     });
12     req.end();
13   });
```

Fig. 2. An example of a dead-listener bug

`min-req-promise` wraps the `http.request` function discussed above, turning its somewhat intricate event-based API with multiple nested listeners into a simpler promise-based API. It exports a function `request`, which returns a promise wrapped around a call to `http.request`. The pending request (an instance of `http.ClientRequest`) is stored in variable `req` (line 4), and a listener function is passed to `http.request` on the same line, which associates it with the `response` event on `req`. Finally, `req.end()` is called on line 12 to dispatch the request. Once a response arrives, the `http` library invokes the listener provided on line 4, passing it a `res` object representing the response, which is an instance of `http.IncomingMessage`. On this object, handlers for three events are installed: `data`, `end` and `timeout`. The first event is emitted whenever a chunk of data from the response arrives, the second when the response has been received in its entirety. For simplicity, we have omitted the handler functions for these two events; the interested reader is referred to the project's GitHub page [45].

The third event, `timeout`, is the problematic one: this event is actually never emitted by `http.IncomingMessage` objects, so the listener on line 10 is dead code. There is a `timeout` event on `http.ClientRequest`, however, so presumably the event should have been registered on `req`, not `res`. We have contacted the author of `min-req-promise`, who has confirmed our analysis of the issue.

Note that there are no compile-time or runtime diagnostics to alert the developer to this problem: not only is it very difficult to infer precise types for variables in JavaScript in general, but there is not even anything semantically wrong with registering a handler for a `timeout` event on `http.ClientRequest`. While the `http` library will never emit this event, client code could do so itself by calling the `emit` method (although in this case it does not). Moreover, since dead-listener bugs do not cause a crash at runtime, they may go undetected for a long time: in the case of `min-req-promise`, the bug had been present since its initial version (released in March 2018).

At present, the only way for a developer to detect this sort of problem is to carefully reason about types and the events they support (as we have done above), or to write extensive unit tests to ensure all events are handled as expected. In the above example, this would require adding a test involving a request that times out, which is an edge case that is easy to overlook.

Clearly, a more automated approach is desirable.

2.3 Automatically detecting dead listeners

We have argued that the dynamic nature of the JavaScript event-driven APIs makes it unrealistic to detect dead listeners at runtime. However, an approach based on static analysis faces the usual dilemma of having to trade off precision against performance: an imprecise analysis is likely to report many false positives, while a very precise analysis will not usually scale to realistic code bases.

Ideally, a static analysis would analyze client code as in Figure 2 along with the implementation of the Node.js standard libraries and any other third-party libraries it depends on, derive a precise model of which types support which events, and then flag dead listeners based on this information. In practice, we know of no static analyzer for JavaScript precise enough to derive such a model that scales to the size and complexity of the libraries involved. As a comparatively benign example, the Node.js `http` package transitively depends on more than 60 modules, for a total of around 20,000 lines of code. While this is quite manageable for, say, type inference or taint tracking, it is out of reach for techniques that precisely model event dispatch, such as that of Madsen et al. [30].

The usual answer is to instead provide the analysis with simplified models of the libraries involved. This is indeed a good approach for frequently used and well-documented packages like `http`, but the modern JavaScript library landscape is vast, with npm alone hosting well over one million packages. While many of these are very rarely used, the number of popular packages is still too large to allow manual modeling, especially since packages tend to go in and out of style quite frequently.

2.4 Approach

Our proposed solution to this dilemma is to turn the size of the JavaScript ecosystem to our advantage in a two-step approach illustrated in Figure 1: first, we mine large amounts of open-source code from GitHub and other hosting platforms for real-world examples of event-listener registrations; then we perform a statistical analysis to determine whether a certain pattern is unusual and hence suggestive of a bug, or whether is common and therefore likely to be correct. This allows us to automatically derive models instead of writing them by hand.

In the next two sections we explain the data mining and classification steps in more detail.

3 DATA MINING

The mining step is implemented as a simple, context and flow-insensitive static analysis that finds event-listener registrations and records them as listener-registration pairs of the form $\langle a, e\rangle$ where a represents the object on which the listener is registered, and e the event for which it is registered.

It is important that both a and e are represented in a code base-independent way to enable the classification step to meaningfully collate results obtained on many different code bases.

For events, this is easy: e is the event name annotated with the emitter package. For instance, `timeout` events on a's rooted in the `http` package are considered to be different from `timeout` events rooted in the `process` package. This is important, as events with the same name in different packages may behave differently.

To represent objects, we use a notion of access paths similar to the one proposed by Mezzetti et al. [32]: starting from an import of a package, the access path records a sequence of property reads, method calls and function parameters that need to be traversed to reach a particular point in the program. More precisely, a conforms to the following grammar:

$$
\begin{array}{rcll}
a & ::= & \mathbf{require}(m) & \text{an import of package } m \\
 & | & a.f & \text{property } f \text{ of an object represented by } a \\
 & | & a() & \text{result of a function represented by } a \\
 & | & a(i) & i\text{th argument of function represented by } a \\
 & | & a_{\mathbf{new}}() & \text{instance of a class represented by } a
\end{array}
$$

Note that access paths are always rooted at a package import, so we can always tell which package any program element derives from.

For instance, in Figure 2, the access path associated with the variable `req` would be **require**(`http`).`request`(), meaning that `req` is initialized to the result of calling the method `request` on the result of importing the `http` module.[1]

The access path of `res`, on the other hand, is **require**(`http`).`request`(1)(0): starting from the import of `http`, we look at a call to `request` as above, but instead of considering the result we look instead at its second argument, [2] which is the listener function on line 4, and then the first argument to that function, which is the variable `res`. As above, the value of the first argument to `request` is not recorded in the access path.

Upon analyzing this snippet of code, we would record three pairs of access paths and events, corresponding to the three explicit event listener registrations:

(1) ⟨**require**(`http`).`request`(1)(0), `data`⟩, corresponds to line 5
(2) ⟨**require**(`http`).`request`(1)(0), `end`⟩, corresponds to line 6
(3) ⟨**require**(`http`).`request`(1)(0), `timeout`⟩, corresponds to line 10

Our approach is based on the assumption that if we collect such pairs on a lot of code, we are likely to see many instances of the first two (correct) pairs, but few instances of the last (incorrect) pair. This is indeed the case: in our experiments (further detailed below) we found 996 instances of the first pair and 898 of the second, but only one of the third.

To detect event-listener registrations, our analysis looks for calls to methods named `on`, `once`, `addListener`, `prependOnceListener` or `prependListener` (the standard Node.js listener registration methods), where the receiver can be represented by an access path, the first argument is a constant string (the event name), and the second argument is a function (the callback).

3.1 Access Path Imprecision

Due to its simplicity, our mining analysis is fairly imprecise. As we will show in Section 6, this does not matter: the statistical analysis in the classification step compensates for much of the imprecision and yields high-quality results. There are two main sources of imprecision: our choice of access paths to represent runtime objects, and the lack of context and flow sensitivity of the analysis.

3.1.1 Imprecision due to access path representation. The formulation of access paths we use is attractive in its simplicity, but it is imprecise because access paths are both *overapproximate* (the same access path may represent many different runtime objects) and *non-canonical* (two different access paths may represent the same runtime object).

As an example of the former, consider again line 4 in Figure 2. This line can equivalently be written like this:

```
const req = http.request(url);
req.on('response', res => { ... });
```

Here, the access path for `res` becomes **require**(`http`).`request`().`on`(1)(0): it is the first parameter of the second argument to `on` invoked on the result of `http.request`. This does *not* record the other arguments to `on`; so, the access path does not include the fact that the first argument to `on` is `response`, and therefore does not contain enough information to allow us to infer the type of `res`. While in actual fact `res` is an `http.IncomingMessage` since the event listener is associated with event `response`, the parameter of an event listener associated with, for example, event `socket` has the same access

[1] Note that the argument to the `request` method is not recorded in the access path; see the Discussion subsection below for more on this point.
[2] We index arguments starting from zero, so the argument at index one is the second argument.

```
var eos = function(stream, opts, callback) {
  // ...
  if ( isRequest( stream)) {
    stream.on('complete', /* ... */ );
    stream.on('abort', /* ... */);
  }
  // ...
```

Fig. 3. Listener registration with type check

path, but its type is `net.Socket`. This means that in some cases we cannot determine event registration correctness based purely on the object's access path: for example, while both `http.IncomingMessage` and `net.Socket` have a `data` event, the former has an `aborted` event that the latter lacks.

As an example of the lack of canonicity of access paths, note that the event registration method `on` returns the emitter event on which it is invoked, so lines 5–10 of Figure 2 could be rewritten as a single statement with three chained listener registrations:

```
res.on('data', /* omitted */)
   .on('end', /* omitted */)
   .on('timeout', () => reject(req));
```

While this does not affect the pair recorded for the first registration, the second becomes ⟨**require**(http).request(1)(0).on(), end⟩. Semantically, **require**(http).request(1)(0).on() and **require**(http).request(1)(0) denote the same set of concrete runtime objects, i.e., they are *aliases*.

Our implementation applies a simple transformation to eliminate chained listener registrations such as the one shown here. In general, cycles in the data-flow graph can give rise to infinitely many access paths that all alias each other, but such cycles are already detected and collapsed by the access-path library used in our implementation.

3.1.2 Access Path Alias Removal. As part of this project, we extended the implementation of access paths available in CodeQL in order to mitigate some of the most common sources of imprecision, as noted from examining our mined data. In particular, we focused on aliases resulting from *chained listener registrations*.

Chained listener registrations are a very common pattern in event-driven JavaScript. The aliasing noted in the rewritten Figure 2 code is true for all chained listener registrations – as explained in the Node.js documentation, the return of an event listener registration is the event emitter itself in order to allow for these chained registrations [18].

In order to remove this source of aliases, we apply the following simplification: access paths representing the return of a listener registration are "condensed" to the base object.

As a concrete example: recall that the access path for `res` is **require**(http).request(1)(0). Then, the access path for `res.on('data', ...)` is **require**(http).request(1)(0).on() – this represents the object on which the listener for `end` is registered. Similarly, the access path for `res.on('data', ...).on('end', ...)` is **require**(http).request(1)(0).on().on().

With our simplification, we recognize that `res.on('data', ...)` and `res.on('data', ...).on('end', ...)` return `res`, and represents these objects with the same base access path **require**(http).request(1)(0). Then, this means that our analysis recognizes that all of these events have listeners registered on the same object.

In our implementation, this is applied as a simple string processing step on reading in the mined data.

3.1.3 Imprecision due to the analysis. The second source of imprecision is the lack of context and flow sensitivity of the analysis, which may cause listener-registration pairs to be reported that can never actually happen at runtime.

```
1 var http = require('http');
2 var server = http.createServer();
3 var client = http.request();
4 client.on('response', function(rsp) {
5     server.emit('response', /* ... */)
6 });
7 server.once('response', function(data) { /* ... */ });
```

Fig. 4. Manual emission of non-standard event

A typical example of this is shown in Figure 3.[3] The function `eos` accepts a variety of streams. Since the `complete` and `abort` events are not emitted by all types of streams, it first checks whether the stream is a request before registering listeners for these two events. Our analysis lacks flow sensitivity, and hence reports `complete` and `abort` event listeners being registered on *all* streams passed as arguments to `eos` that do *not* support these events (in this particular example, streams of type `http.IncomingMessage`).

Finally, note that our mining analysis does not account for code that emits the events. This means that it may report a pair $\langle a, e \rangle$ that is, in general, incorrect because a is a library API that does not emit event e, but happens to be correct for a particular code base, because that code base manually emits e on a.

For example, consider Figure 4.[4] On line 7 we see a listener to `response` registered on the result of a call to `http.createServer()`, which is an object of type `http.Server`. According to the API documentation of the `http` library, `http.Server` does not emit the `response` event. However, in this instance the client application itself emits a `response` on the server object (line 5), and hence the `response` listener is not dead. We could improve our analysis to suppress listener-registration pairs for which it sees a manual emit, but we decided against doing this in the interest of simplicity.

These various sources of imprecision can make the data produced by the mining step somewhat noisy, but the classification step mitigates this problem: its input is collected from a large set of code bases, the majority of which do not use tricky idioms like these.

4 CLASSIFICATION

Once we have collected a large corpus of listener-registration pairs we want to categorize them to identify pairs that are likely to correspond to API misuses. We first describe the general intuition behind our approach, which we make precise in a statistical model. We then explain how the statistical model is applied to identify potentially buggy pairs, and finally introduce a refinement to avoid miscategorizations.

4.1 General intuition

As argued above, if the analyzed corpus is big enough, buggy listener-registration pairs are likely to be relatively rare. However, the converse is not true: there are two situations where a rare listener-registration pair is not indicative of a problem.

Rare event emitter. If an event emitter is infrequently used, for example because it belongs to a *rarely-used API* or *custom API extension*, then we will not see many listener registrations on this emitter overall. In particular, any listener-registration pair involving this emitter will appear to be rare (when compared to the entire set of listener-registration pairs collected).

[3] Adapted from the `mafintosh/end-of-stream` project on GitHub
[4] Adapted from the `strongloop/strong-pm` project on GitHub

As an example, consider the following listener registration from the GitHub project `martindale/soundtrack.io`:

```
req.spotify.get(url).on('complete', /* omitted */)
```

Here, `req` is an instance of `http.ClientRequest`. Objects of this class do not normally have a `spotify` property. This is a custom property added by `soundtrack.io` for interacting with the Spotify API.[5] Consequently, we see the access path of this registration very infrequently; we only encountered it twice in our evaluation.

Further study of the source code reveals that `req.spotify.get` does, indeed, return an event emitter that supports the `complete` event, so this listener registration is correct.

Rare event name. The Node.js event API allows client code to emit and register listeners for *custom events*. Hence, a listener-registration pair may be infrequent simply because the event is a custom event only used in one particular code base.

For example, the test suite of the `emitter-listener` npm package [36] uses a custom event `test` on `http.ServerResponse` objects. This is encountered three times in this particular code base, and all three instances correspond to correct usages of the custom event. We would not expect this pair to appear anywhere else (and indeed we did not find any other instances in our evaluation), but in spite of its rarity it is a correct pair.

To avoid the above two situations, it is not enough to consider the rarity of the pair when compared to all other pairs. We want to only consider a listener-registration pair $\langle a, e \rangle$ as unusual (and hence potentially buggy) if *both* of the following hold:

(1) e is a rare event for a;
(2) a is a rare access path for e.

The first condition excludes rare event emitters, as in the example from `martindale/soundtrack.io` above: the access path only occurs in two listener-registration pairs, one of which registers the `complete` event. Hence `complete` appears in 50% of all pairs involving the access path, meaning that it is (intuitively) not a rare event for the access path.

The second condition excludes rare event names, as in the example from `emitter-listener`: the `test` event only occurs in three listener-registration pairs,[6] one of which registers it on **require**(`http`).ServerResponse$_{\mathbf{new}}$(). Hence this access path appears in 33% of all pairs involving the `test` event, meaning that it is (intuitively) not a rare access path for the event.

We now develop a statistical model to make our intuitive notion of rarity rigorous and effectively computable.

4.2 Statistical model

Our model is parameterized over four threshold values, two *rarity thresholds* p_a and p_e, and two *confidence thresholds* p_{ca} and p_{ce}, all of which range between 0 and 1 (as they represent probabilities).

The rarity threshold p_a determines when we consider an access path to be rare for an event, and vice versa for p_e. For example, $p_a = 0.05$ means that we consider an access path a rare for an event e if it occurs in less than 5% of all listener registrations involving e. Equivalently, we can word this as: a is rare for e if the probability that for an arbitrary pair $\langle a', e \rangle$ we find $a' = a$ is less than 0.05.

Note that we cannot measure this probability directly, since the data set we base our model on only covers a small fraction of the universe of all existing or possible JavaScript code. This is where the confidence thresholds come in:

[5]Recall that in JavaScript the properties of an object are not fixed; properties can be added, overwritten, and deleted dynamically.
[6]Recall that we disambiguate event names based on the root package of the access path we see them registered on. While many packages have a `test` event, in this case we are only interested in `test` events related to the `http` package.

we use a confidence test to determine, for a given listener-registration pair $\langle a, e\rangle$, how confident we are based on our limited data set that a is a rare access path for e. This confidence has to be within the limit set by the threshold p_{ca} before we are willing to accept a is actually rare for e. Again, p_{ce} is the dual threshold, specifying the required level of confidence for us to recognize event e as rare for an access path a.

4.3 Applying the model

We will now explain how our statistical model uses these thresholds to determine if a listener-registration pair $\langle a, e\rangle$ is unusual (and that it should therefore be categorized as a potential API misuse).

First, we want to determine whether a is rare for e. Let p be the probability that, for an arbitrary pair $\langle a', e\rangle$, we find $a' = a$. We want to test whether, based on our data set, it is highly likely that p is below the rarity threshold p_a. In the manner of a classical hypothesis test, we will actually test the converse: whether it is highly *un*likely that the probability p is *greater than* (or equal to) the threshold p_a. More formally, if the likelihood of p being greater than or equal to p_a is *below* our confidence threshold p_{ca}, then we conclude that p must, in fact, be less than p_a, and hence that a is rare for e. This is why we will generally want to choose small values for our confidence thresholds.

As explained above, we cannot directly measure p, but we can measure n_e, the number pairs $\langle a', e\rangle$ we observed, and k, the number of pairs among these for which $a' = a$. For example, for the pair ⟨**require**(`http`).`request`(1)(0), `timeout`⟩ corresponding to the bug in Figure 2 we have $n_e = 216$ and $k = 2$: the `timeout` event occurs in 216 pairs, but only twice with the access path **require**(`http`).`request`(1)(0). This outcome is consistent with, eg., $p = 0.01$, but makes higher values like $p > 0.05$ seem unlikely.

To make this intuition rigorous, we model the probability distribution of the number of times an access path appears together with a particular event using the binomial distribution function. Then, we want to determine if the likelihood of observing no more than k occurrences of the pair $\langle a, e\rangle$ among n_e pairs involving event e is less than the confidence threshold p_{ca} (under the assumption that the probability p that for an arbitrary pair $\langle a', e\rangle$ we find $a' = a$ is greater than or equal to p_a), and so we use the binary cumulative distribution function BCDF [10] to test:

$$\mathrm{BCDF}(k, n_e, p_a) < p_{ca}$$

Plugging in our example values from above, we get $\mathrm{BCDF}(2, 216, 0.05) \approx 0.001$, meaning that based on our observations the likelihood of p being greater than 0.05 is 0.1%. Turning this statement around, we are 99.9% certain that a occurs in 5% or less of all access pairs involving e. Now, to conclude that a is indeed rare for e (with our rarity threshold $p_a = 0.05$), we need this 99.9% certainty to satisfy the chosen confidence threshold p_{ca}. If, for example, we chose a $p_{ca} = 0.05$, the confidence threshold would be 95% and so we *would* conclude that a is rare for e.

To test whether e is also rare for a we follow the same approach with thresholds p_e and p_{ce} instead. Putting it all together, then, we consider our listener-registration pair $\langle a, e\rangle$ to be rare if both tests succeed, that is, if the following condition holds:

$$\mathrm{BCDF}(k, n_a, p_e) < p_{ce} \ \wedge \ \mathrm{BCDF}(k, n_e, p_a) < p_{ca}$$

4.4 Refining the model

Applying this condition in practice, we noticed one particular scenario where it led to miscategorizations: if for an event e there are many pairs $\langle a, e\rangle$, but each individual pair occurs infrequently, we will end up categorizing all access paths a for this event as rare. This pattern arises, for instance, with custom events used in tests.

As a concrete example, there are 522 $\langle a, e\rangle$ pairs registering a listener for the `doge` event on an a rooted at the npm package `socket.io-client`. This nonsensical event name is commonly used for a placeholder or test event – this is reflected in the data, as we see that these 522 pairs involve 520 unique paths, 519 of which occur in exactly one pair. In other words, the usage of `doge` follows no discernible pattern in our data.

For one of the pairs $\langle a, \text{doge}\rangle$ with an access path a that only occurs once and a rarity threshold p_a of 0.01, we get $\text{BCDF}(1, 522, 0.01) \approx 0.03$, so we would confidently conclude (with 97% confidence) that a is rare for `doge` and might then label it as unusual. This is clearly not desirable; instead, we should not conclude anything about this pair, since our data is too sparse.

We encode this into our model by changing our occurrence count k to not only count occurrences of the pair $\langle a, e\rangle$, but also occurrences of pairs $\langle a', e\rangle$ where a' appears together with e as often or less often than a.

Formally, we write $k_e(a)$ for the number of times the pair $\langle a, e\rangle$ occurs in our data (which we wrote as k above), and then define

$$k_e(\lceil a \rceil) = \sum \{k_e(a') \mid 0 < k_e(a') \leq k_e(a)\}$$

Intuitively, this means that we are now not only taking into account the absolute number of times we see a together with e, but also how that number compares to that of other as (on the same e). For example, for the 519 access paths that only appear once together with `doge`, we now have $k_e(\lceil a \rceil) = 519$, making them very unlikely to be considered rare.

Defining $k_a(\lceil e \rceil)$ symmetrically as the number of occurrences of pairs $\langle a, e'\rangle$ where e' appears together with a as often or less often than e, we refine our overall condition for a pair $\langle a, e\rangle$ being unusual as follows:

$$\text{BCDF}(k_a(\lceil e \rceil), n_a, p_e) < p_{ce} \ \wedge \ \text{BCDF}(k_e(\lceil a \rceil), n_e, p_a) < p_{ca}$$

In particular, our single-occurrence access paths above now fail the second condition since $\text{BCDF}(k_e(\lceil a \rceil), 522, 0.01) = \text{BCDF}(519, 522, 0.01) \approx 1$, that is, we are almost 100% confident that these access paths do not meet the rarity threshold of 0.01.

It should be noted, however, that this formulation does result in more false negatives: if any of these access paths is actually incorrect, they will no longer be flagged. Since we are mostly interested in automated bug detection, we are willing to trade false positives for false negatives.

5 IMPLEMENTATION

In this section we provide some details on the implementation of the two stages of our approach.

For the mining stage, we implemented a static analysis in QL [8] for identifying event registrations in JavaScript. Extensive libraries for writing static analyzers in QL are available as part of CodeQL [23], including, in particular, an implementation of access paths, making it an ideal tool for our purposes. Moreover, by writing our analysis in QL we were able to leverage LGTM.com [22], a cloud-based analysis platform that, at the time of writing, makes over 130,000 open-source code bases from GitHub, Bitbucket, Gitlab and other hosting providers available for analysis. Out of these, around 127,500 contain at least some JavaScript code, which we use as the basis of our evaluation in Section 6.

For the categorization stage, we implemented the approach detailed in Section 4 in Python, using the pandas library [13], and the SciPy library [14] for the statistical computations.

6 EVALUATION

To evaluate the practical usefulness of our approach, we analyze uses of event-based APIs collected from a large collection of open-source JavaScript code bases and assess the results quantitatively and qualitatively with the following research questions:

RQ1. *Impact of configuration parameters on precision/recall*: How do precision/recall change as the configuration parameters vary?

RQ2. *Impact of training set selection on precision/recall*: How do precision/recall change as the training set selection varies?

RQ3. *Impact of training set size on precision/recall*: How do precision/recall change as the training set size varies?

RQ4. *Utility of results*: Does our approach identify practically relevant mistakes?

RQ5. *Practicality*: Is our approach practical in terms of performance/resources?

6.1 Generation of labeled pairs for model validation

In order to evaluate our model, we need to determine the accuracy of the classification of pairs. In practice, the sheer number of pairs we are dealing with makes exhaustive manual classification impossible. Instead, we semi-automatically classified a set of pairs for those 18 packages for which our mining phase found the largest number of event registrations.[7]

For each of these packages, we studied the API documentation and made lists of the events emitted by and access paths that correspond to each API type. These access paths, paired with the events their types are known to emit, forms the set of correct API uses. Then, we created the corresponding list of all the events that get emitted by *other* types, and similarly use this information to create a set of incorrect API uses.

For example, from looking at the `http` API documentation, we infer that objects of type `http.Server` emit event `connection`. So, access paths for objects of type `http.Server` paired with `connection` are considered correct. We also see that `http.Socket` objects do *not* emit `connection` events. Therefore, access paths for objects of type `http.Socket` paired with `connection` are considered incorrect. As discussed in Section 3.1, some access paths are imprecise. These imprecise paths paired with every event from the API form the list of *imprecise pairs*.

We codified this list of types and corresponding events in a script to generate lists of these correct, incorrect, and imprecise pairs from our mined data. In practice, this amounts to labelling all the pairs in our mined data that correspond to one of these sets. The generated model for `http` is included in the appendix, as an example.[8]

We now address each of the research questions in turn.

6.2 RQ1: Impact of configuration parameters on precision/recall

Since our goal is to automatically find dead listener patterns, our approach has to achieve two things to be practically useful: it should flag as many incorrect listener-registration pairs as possible, while at the same time flagging as few correct pairs as possible. In other words, we should maximize for both recall *and* precision.

How well our approach achieves these goals depends on the parameters of our statistical model, so we systematically explore the space of configurations to find one that maximizes recall while maintaining an acceptable precision rate (defined as 90% in accordance with the literature [43, 44]).

[7] `http`, `net`, `fs`, `process`, `child_process`, `https`, `socket.io`, `socket.io-client`, `stream`, `readable-stream`, `events`, `cluster`, `zlib`, `ws`, `readline`, `http2`, `repl`, `tls`

[8] All the models, and the model-generating script is available at: https://github.com/emarteca/JSEventAPIModelling.

First, we collected a corpus of listener-registration pairs by running our mining analysis on all 127,531 JavaScript projects currently available on LGTM.com. We found a total of 532,004 $\langle a, e \rangle$ pairs (160,195 unique), from 35,757 projects. The remaining projects did not use event-based APIs recognized by our analysis.

For each configuration of the four parameters p_a, p_e, p_{ca} and p_{ce} we can then run our categorization to find unusual (and hence possibly incorrect) listener-registration pairs. Ideally, we would then manually inspect each pair $\langle a, e \rangle$ flagged by the categorization to label it as either a true positive (that is, emitters represented by a really do not emit event e, so any listener for e on a will be dead) or a false positive (at least some emitters represented by a do emit event e). In particular, we count pairs where a is too imprecise to determine the runtime type of the emitter as false positives.

Since this is infeasible in practice, we instead use the generated set of labelled pairs described in Section 6.1 to determine the precision and recall of the model at each configuration. From these generated models we inferred 959 pairs as being correct API uses, 4323 pairs as having an imprecise access path, and 399 as incorrect API uses, for a total of 5681 of labeled pairs. True positives are pairs that the model classifies as being a bug pattern that are also included in the labelled incorrect set. False positives are pairs that the model classifies as being a bug pattern that are in the labelled correct set. We also consider pairs that the model classifies as being a bug pattern that are in the labelled imprecise set as false positives, to report the most pessimistic results for our technique. We also report the *unclassified pairs*: these are pairs the model classifies as being a bug pattern that are not in our labelled data at all. However we do not include these pairs in our precision/recall computations, since we cannot automatically validate their correctness.

Next, we need to choose parameter values to test. For the rarity threshold parameters p_a and p_e, we chose values from the set $\{0.005, 0.01, 0.02, 0.03, 0.04, 0.05, 0.1, 0.25\}$. A value of $p_a = 0.005$, for instance, means that we consider an access path to be rare for an event if it occurs in less than 0.5% of all pairs with this event.

For the confidence threshold parameters we chose values from the set $\{0.005, 0.01, 0.02, 0.03, 0.04, 0.05, 0.1, 1\}$. A value of $p_{ca} = 0.005$, for instance, means that we want to be 99.5% sure that an a is rare for an e before categorizing it as rare. The extreme value of $p_a = 1$ has the effect of categorizing every a as rare for e, thereby reducing the model to just checking whether events are rare for access paths (and vice versa). This allows us to test the sensitivity of our model to the rarity of access paths and events individually.

Altogether, this gives us a space of 4096 configurations, which we explore exhaustively, computing precision and recall relative to the labeled data set for each of them. Figure 5 shows the results of this experiment. Unsurprisingly, there is an inverse correlation between the recall and precision: configurations that flag many pairs as being unusual have many true positives, but also many false positives. Hence it is not meaningful to optimize either metric in isolation.

Instead, we want to concentrate on the *Pareto front* [31, Chapter 16], that is, the set of configurations for which there is no other solution that is better on both metrics (the red line in Figure 5): a configuration is in the Pareto front if there is no configuration with the same (or higher) precision that has a higher recall.

Altogether, there are eight configurations on the Pareto front with precision of 80% or above, as detailed in Table 1. For each configuration we show the values of the four parameters, the precision and recall, and the unique true-positive, false-positive, and unclassified counts (the unclassified pairs are those flagged as incorrect that are not in our labelled data). We also show the number of times these true positives occur in our entire data set (roughly speaking, this is the number of potential bugs the configuration finds) and the number of projects they occur in. For example, the first row reads as follows: a parameter configuration of p_a, p_e, p_{ca}, and p_{ce} as 5%, 5%, 2%, and 10% respectively, results in a precision of 100% and recall of 3.0% over the labelled data set. This corresponds to 12 true positives (#TP), no false positives (#FP), and no unclassified pairs (#UP); these true positive pairs occur 23 times in the mined data (Occ TP), across 22 projects (#Proj).

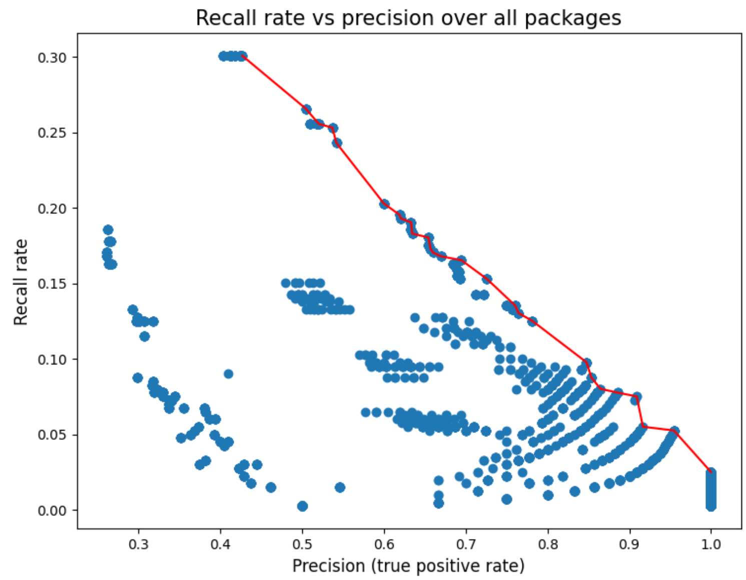

Fig. 5. Precision and recall for all configurations (blue dots); Pareto front in red

Configuration	**Results**						
$(p_a, p_e, p_{ca}, p_{ce})$	% Precision	% Recall	# TP	# FP	# UP	Occ TP	# Proj
(0.05, 0.05, 0.02, 0.1)	100.0	3.0	12	0	0	23	22
(0.1, 0.05, 0.05, 0.1)	95.8	5.8	23	1	0	57	48
(0.1, 0.05, 0.1, 0.1)	92.3	6.0	24	2	0	58	49
(0.1, 0.1, 0.03, 0.01)	90.9	7.5	30	3	1	75	64
(0.25, 0.04, 0.01, 0.005)	88.6	7.8	31	4	3	77	61
(0.25, 0.05, 0.01, 0.01)	86.5	8.0	32	5	3	79	63
(0.25, 0.01, 1, 0.04)	85.4	8.8	35	6	9	48	36
(0.25, 0.01, 1, 0.1)	84.8	9.8	39	7	12	55	41

Table 1. Configurations with ≥80% recall; optimal configuration highlighted in gray.

To answer RQ1, then, we found that there are indeed configurations with more than 90% precision. The fourth row represents the configuration we consider optimal: This is the configuration that yields the highest recall for a precision over 90%. The rarity thresholds p_a and p_e are 10% and 10%, and the confidence thresholds p_{ca} and p_{ce} are 3% and 1%, respectively. Over the labeled data set, this configuration yields three false positives and 30 true positives, for a precision of 90.9%. The true-positive pairs occur 75 times in total across 64 projects. All the false positives for this configuration were cases where the access path is overly imprecise.

6.3 RQ2: Impact of training set selection on precision/recall

The configuration we identified as optimal in RQ1 performs very well on our labelled data set, but of course this does not imply that it would do as well on another data set. In order to address this concern without having to manually label even

Round	Configuration	Training			Validation		
	$(p_a, p_e, p_{ca}, p_{ce})$	% Precision	% Recall	# TP	% Precision	% Recall	# TP
0	(0.25, 0.04, 0.01, 0.005)	90.6	8.1	29	87.5	5.0	2
1	(0.25, 0.04, 0.01, 0.005)	91.2	8.6	31	75.0	7.5	3
2	(0.1, 0.1, 0.03, 0.01)	92.0	6.4	23	87.5	17.5	7
3	(0.1, 0.05, 0.1, 0.1)	90.9	5.6	20	100.0	10.0	4
4	(0.1, 0.05, 0.1, 0.1)	91.7	6.1	22	100.0	5.0	2
5	(0.1, 0.1, 0.04, 0.01)	90.3	7.8	28	75.0	5.0	2
6	(0.25, 0.04, 0.01, 0.005)	90.0	7.5	27	80.0	10.0	4
7	(0.1, 0.1, 0.03, 0.02)	90.6	8.1	29	87.5	5.0	2
8	(0.1, 0.1, 0.03, 0.01)	90.3	7.8	28	100.0	5.0	2
9	(0.1, 0.1, 0.03, 0.02)	90.3	7.8	28	100.0	7.5	3

Table 2. Outcomes of cross-validation experiment

more pairs, we conducted a 10-fold cross-validation experiment. We divided our data into 10 random partitions. Then, we determine the best configuration (i.e., the highest recall with at least 90% precision) over nine of these partitions (the training set) and validate them on the remaining partition. We repeat this procedure ten times, once for each of the partitions as the validation set.

The results of the experiment are shown in Table 2. Each row represents the results of using a group of nine partitions as the training set and the remaining partition as the validation set. The second column shows the optimal configuration over the training set. Columns 3-5 show the precision, recall, and absolute true positive count on the training set, while columns 6-8 show the same on the validation set. For example, the first row reads as follows: in the first round the optimal configuration on the training set was $p_a = 0.25$, $p_e = 0.04$, $p_{ca} = 0.01$, $p_{ce} = 0.005$, which achieved a 90.6% precision with 8.1% recall, finding 29 true positive results. On the validation set, that same configuration resulted in a precision of 87.5% and recall of 5.0%, with 2 true positive results.

We see consistent results with the cross-validation experiment. Concretely: across the 10 rounds of the experiment, in the training set we see an average precision of 90.8% (standard deviation 0.7%) and an average recall of 7.4% (standard deviation 1.0%). Then, in the validation set we see an average precision of 86.3% (standard deviation 10.3%) and an average recall of 7.8% (standard deviation 4.0%). From this we see that not only is the quality of results consistent between training runs, but that it also results in consistent results on the validation sets.

Looking at the configurations determined to be optimal, we see a high occurrence rate of each of our parameters determined optimal over the whole set. Concretely: our optimal $p_a = 0.1$ is found in 7 runs and $p_a = 0.25$ (resulting in a precision of 88.6% over the whole set) is found in the other 3. Similarly, our optimal $p_e = 0.1$ is found in 5 runs, $p_{ca} = 0.03$ in 4 runs, and $p_{ce} = 0.01$ in 3 runs. In conclusion, the choice of training set does not substantially affect the choice of optimal configuration.

6.4 RQ3: Impact of training set size on precision/recall

Having shown that the *selection* of the training set does not matter much, we will now explore the effect of the *size* of the training set: how well does our model perform when trained over smaller data sets?

In order to test this, we designed an experiment where we randomly sampled a given percentage of the data, and then determined the optimal configuration on this subset. As before, we define "optimal" to mean the configuration with the highest recall that achieves a precision of at least 90%. Then, we take this configuration and report the precision/recall it achieves over the whole data set. We repeat this process 10 times for each percentage, and test this on samples of 2%, 5%, 10%, 25%, and 50% of the total data. For each of these sample percentage, we report the average (harmonic mean) precision and recall computed over all 10 iterations.

Table 3 presents the results of this experiment, with one row per sampling of a given percentage. For example, the first row can be read as follows: for the first random sampling of 2% of the data, the optimal configuration is $(0.25, 0.01, 1, 0.1)$. This achieves a precision of 100% and a recall of 50% over the 2% subset, and a precision of 84.8% and a recall of 9.8% over the whole dataset.

Considering only average precision and recall over the whole dataset for the moment, we can see two interesting trends: when training on 2% of the data, precision over the whole dataset is fairly low at 82.4%, and so is recall at 6.9%. As the amount of training data increases, precision increases as well, reaching 91% for 25% of the data. Recall, on the other hand, at first *decreases*, hitting a low of 3.6% on 25% of the data, before slightly increasing again to 3.9% at 50%. (Compare this to the 7.5% recall reported in RQ1 when training on all 100% of the data.) This suggests that stable precision can already be achieved with relatively little training data, but more data is needed to improve recall.

Looking more closely at the individual configurations determined to be optimal, we see that for 2% these configurations always have $p_{ca} = 1$, meaning that *all* access paths are considered rare, and our categorization is entirely based on p_e and p_{ce}. As discussed in Section 4, this means that our analysis will not handle custom events very well, and end up spuriously flagging custom events on common access paths as bugs. This, it turn, leads to very unstable precision, where 100% precision on the training set dwindles to an average of only 82.4% over the whole data set. At higher percentages, configurations with $p_{ca} = 1$ become increasingly rare, with only one left at 50%: as more data becomes available, our approach learns to deal with custom events better, and precision becomes more stable.

Finally, note that recall on the subset (third column) varies wildly between different samplings at smaller percentages. This is because there are few labeled pairs in those datasets, meaning that missing one or two pairs will significantly affect recall.

In conclusion, with small training sets our model converges to configurations that do not account for custom events, leading to unstable precision and low recall. As more data is provided, precision becomes better and more stable, while recall is still way below what we can achieve by training on 100% of the dataset.

6.5 RQ4: Utility of results

To qualitatively assess the usefulness of our approach, we conducted a study involving finding bugs in open-source projects.

In this experiment, we examined occurrences of listener-registration pairs that were classified as erroneous by the statistical model, and manually examined the code from which the pair originated. For those results that appeared to flag real bugs in the code, we submitted issues to report them to the developers. Altogether, we reported 30 issues across 25 different GitHub projects, 7 of which have been confirmed by the developers as bugs,[9] one where the developers did not remember what the code was supposed to do, and 2 that were false positives (due to manually emitting the event elsewhere in the project). We did not yet receive a response for the remaining 20 issues. Links to all reported issues are included in the appendix.

Figure 6 shows a simplified version of one of the acknowledged bugs from the project `abalabahaha/eris`, a Node.js wrapper for interfacing with Discord. Here we see the `req` variable created from a call to `http.request`, which returns an object of type `http.ClientRequest`. However, by examining the `http` API documentation, we see that `aborted` is

[9] Two have been addressed, [1] and [2].

2% of data sampled for subset

		On subset		On whole set	
Iter	Optimal config	% Precision	% Recall	% Precision	% Recall
1	(0.25, 0.01, 1, 0.1)	100.0	50.0	84.8	9.8
2	(0.25, 0.01, 1, 0.1)	100.0	25.0	84.8	9.8
3	(0.25, 0.02, 1, 0.005)	100.0	71.4	78.1	12.5
4	(0.25, 0.005, 1, 0.1)	100.0	12.5	84.2	4.0
5	(0.25, 0.02, 1, 0.005)	100.0	42.9	78.1	12.5
6	(0.25, 0.005, 1, 0.1)	100.0	46.2	84.2	4.0
7	(0.25, 0.03, 1, 0.005)	100.0	41.7	71.8	15.3
8	(0.25, 0.01, 1, 0.05)	100.0	50.0	84.1	9.3
9	(0.25, 0.005, 1, 0.05)	100.0	14.3	93.3	3.5
10	(0.25, 0.01, 1, 0.02)	100.0	37.5	83.8	7.8
			Average (harmean):	82.4	6.9

5% of data sampled for subset

Iter	Optimal config	% Precision	% Recall	% Precision	% Recall
1	(0.25, 0.03, 0.005, 0.05)	100.0	25.0	87.5	7.0
2	(0.25, 0.04, 0.1, 0.005)	90.0	30.8	82.9	8.5
3	(0.1, 0.1, 0.02, 0.05)	100.0	21.1	81.6	7.8
4	(0.25, 0.02, 1, 0.03)	90.0	37.5	76.4	13.8
5	(0.04, 0.02, 0.005, 0.05)	100.0	7.1	100.0	1.3
6	(0.25, 0.02, 1, 0.1)	100.0	13.6	81.3	6.5
7	(0.25, 0.03, 0.05, 0.03)	100.0	10.0	81.1	7.5
8	(0.25, 0.005, 1, 0.01)	100.0	29.4	90.0	4.8
9	(0.25, 0.02, 1, 0.05)	100.0	36.4	80.6	6.3
10	(0.25, 0.02, 1, 0.1)	100.0	22.7	81.3	6.5
			Average (harmean):	83.8	4.8

10% of data sampled for subset

Iter	Optimal config	% Precision	% Recall	% Precision	% Recall
1	(0.1, 0.05, 0.05, 0.005)	100.0	5.1	95.2	5.0
2	(0.25, 0.005, 1, 0.05)	100.0	29.0	93.3	3.5
3	(0.1, 0.04, 0.05, 0.01	100.0	15.8	95.0	4.8
4	(0.25, 0.02, 0.1, 0.1)	100.0	13.5	81.3	6.5
5	(0.05, 0.1, 0.01, 0.01	92.9	33.3	90.5	4.8
6	(0.25, 0.005, 1, 0.03)	90.0	22.5	92.9	3.3
7	(0.25, 0.005, 1, 0.03)	100.0	31.3	92.9	3.3
8	(0.25, 0.005, 1, 0.1)	100.0	28.1	84.2	4.0
9	(0.25, 0.02, 0.1, 0.1)	100.0	13.9	81.3	6.5
10	(0.25, 0.03, 0.05, 0.03)	90.0	20.5	81.1	7.5
			Average (harmean):	88.4	4.5

25% of data sampled for subset

Iter	Optimal config	% Precision	% Recall	% Precision	% Recall
1	(0.04, 0.05, 0.1, 0.03)	92.3	10.4	90.5	4.8
2	(0.1, 0.01, 0.02, 0.05)	90.0	8.7	90.0	2.3
3	(0.25, 0.04, 0.1, 0.02)	92.6	21.6	81.0	8.5
4	(0.03, 0.03, 0.05, 0.04)	100.0	6.9	100.0	2.0
5	(0.25, 0.005, 1, 0.03)	90.0	12.0	92.9	3.3
6	(0.25, 0.005, 1, 0.04)	100.0	12.6	92.9	3.3
7	(0.25, 0.005, 1, 0.05)	100.0	15.2	93.3	3.5
8	(0.25, 0.03, 0.02, 0.02)	91.7	11.8	84.4	6.8
9	(0.25, 0.005, 1, 0.03)	90.9	9.8	92.9	3.3
10	(0.1, 0.05, 0.05, 0.05)	90.9	8.5	95.7	5.5
			Average (harmean):	91.0	3.6

50% of data sampled for subset

Iter	Optimal config	% Precision	% Recall	% Precision	% Recall
1	(0.05, 0.05, 0.1, 0.05)	100.0	7.0	92.9	3.3
2	(0.05, 0.05, 0.05, 0.1)	93.8	7.9	92.9	3.3
3	(0.02, 0.05, 0.1, 0.02)	100.0	5.9	100.0	1.8
4	(0.04, 0.05, 0.01, 0.02)	92.9	6.6	100.0	2.8
5	(0.25, 0.03, 0.02, 0.04)	94.4	8.1	84.8	7.0
6	(0.25, 0.04, 0.01, 0.005)	90.0	13.2	88.6	7.8
7	(0.25, 0.01, 1, 0.04)	92.6	12.6	85.4	8.8
8	(0.05, 0.05, 0.05, 0.04)	92.9	7.2	92.3	3.0
9	(0.25, 0.02, 0.1, 0.005)	91.3	10.2	81.5	5.5
10	(0.25, 0.04, 0.02, 0.005)	90.5	9.6	86.1	7.8
			Average (harmean):	91.0	3.9

Table 3. Optimal configurations over smaller percentages of the data, and the corresponding precision/recall over the whole dataset

```
1 const HTTPS = require("https");
2
3 request(method, url, auth, body, file, _route, short) {
4   const req = HTTPS.request( /* ... */ )
5   req.once("abort", () => { /* ... */ }
6   ).once("aborted", () => {/* ... */ }
7   );
8   req.once("response", (resp) => { /* ... */ });
9 }
```

Fig. 6. Condensed version of error in `abalabahaha/eris`

an event emitted by `http.IncomingMessage`, and not `http.ClientRequest`. The developers confirmed this as a bug and fixed it by registering the listener on `resp` instead (line 8), which had been the original intention.

The full list of bugs reported is in the appendix.

6.6 RQ5: Practicality

Here, we discuss performance and resource requirements.

Data mining and classification: Our approach involves mining and classifying listener registration pairs from a large number of projects. The data mining step requires about 404 hours of compute time for the 127,531 projects in our data set. Since LGTM.com runs queries concurrently, this step was completed in about two days. The categorization stage is much faster: classifying the pairs for a given configuration takes only 35 to 40 seconds on commodity hardware. We expect these steps to be applied infrequently as event-driven APIs tend to evolve slowly, and our experimental results suggest that the set of optimal analysis thresholds is fairly stable.

Per-project costs: Once an API model has been constructed, it can be used for a variety of purposes, e.g., in a bug-detection tool that flags uses of event-driven APIs that are likely to be incorrect, or in an IDE plugin for smart completion. Running our mining analysis on a single JavaScript project is quite fast: for 52% of all projects in our data set, the analysis takes ten seconds or less, with another 45% taking between ten seconds and a minute. There are only 151 projects (0.1%) for which the analysis takes more than ten minutes.

We consider these results to be encouraging as, while the upfront cost of constructing an API model is quite high, our experimental results suggest that the per-project costs are sufficiently low to allow integration of our tools in a realistic development/build workflow.

6.7 Threats to Validity

We are aware of several potential threats to validity.

Our results depend on the set of code bases we mine, which may not be representative. However, we simply used the set of *all* JavaScript projects on LGTM.com, which includes many popular open-source projects, and projects added by users of LGTM.com. These code bases were not specifically selected for this project, and provide a reasonable sample of real-world JavaScript code.

Our experimental measurements of precision and recall are based on a relatively small set of listener-registration pairs that we semi-automatically labeled as correct or incorrect (5681 out of 160,195 unique pairs) and might not generalize beyond this set. Exhaustively labeling all pairs was infeasible, so we focused on the most popular packages we encountered, to ensure that our results are relevant for widely-used APIs. Cross-validation showed that the choice of optimal configuration does not crucially depend on the chosen training data set.

Our labeled data set is itself biased in that it contains a relatively small number of pairs labeled as incorrect (399 out of 5681). This affects the accuracy of our reported precision since we are much more likely to find that a pair flagged by the model is actually correct (and hence a false positive) than incorrect (and hence a true positive). Consequently, our reported precision *underestimates* the actual precision. Remedying this imbalance would only improve our results.

The values chosen for the parameters of our statistical model obviously greatly influences the quality of results. However, our evaluation considered a large number of different combinations, over which we determined the optimal configuration for a particular set of conditions (here, for a precision of at least 90%). Moreover, a cross-validation experiment revealed the configuration parameters to be quite stable across subsets of the data.

Finally, the static analysis used in the mining phase is relatively simple and imprecise, e.g., due to inherent imprecision of the access path representation. Our evaluation accounted for this by considering all pairs involving imprecise access paths to be false positives. A more sophisticated analysis using more precise access paths would also increase the precision of our model.

6.8 Learning lost events

At first glance, it seems that we could apply this same learning approach to the dual problem of finding bug patterns in lost events [30], those events which are emitted but never listened for. We modified our static analysis to identify event emissions instead of event registrations, and reran the data mining to collect information on this dual problem, across the same set of projects. From this analysis, we mined a total of 22,900 $\langle \textit{emitter access path}, e \rangle$ pairs (10,432 unique).

From this data, we determined that the learning approach cannot be effectively used to identify bug patterns in lost events, since the vast majority of events emitted in projects are *custom events*. The use of a custom event is local to the project it appears in, and so patterns observed in other projects cannot be used to learn about its proper use. We discussed this in Section 4.1 with respect to listener registrations, but the same logic applies to event emissions.

For the remainder of this section, we discuss some details about the data we mined on events emitted.

Base event emitter. Of the 22,900 $\langle \texttt{emitter access path}, e \rangle$ pairs we mined, we observed that 5248 of these have the emitter access path $\mathbf{require}(\texttt{events}).\texttt{EventEmitter}_{\texttt{new}}()$ and 1220 have the emitter access path $\mathbf{require}(\texttt{events})_{\texttt{new}}()$. These are the first and second most frequent emitter access paths in our mined data respectively.

In code, these access paths correspond to `new require('events').EventEmitter()` and `new require('events')()` respectively. Looking at the documentation of the EventEmitter class[10], we see that these are aliases, as the EventEmitter class is the default export of the `events` package. Looking at the same documentation, we see that there are only 2 events that make up this API: `newListener` and `removeListener`. Any other events emitted on objects of this type are therefore custom events. With our data, we see that 6466 of these 6468 pairs emit a custom event on objects of the base event emitter class.

We examined what users actually do with the custom events on base event emitters. From our exploration, a few common patterns of how users build their own custom event infrastructures on top of the base EventEmitter class could be observed; we include examples of these below.

Developers often create custom EventEmitter classes extending the base EventEmitter class in a classic object-oriented style. Consider the following demonstrative example, condensed from the update manager class in `vscode`.

```
export class UpdateManager extends events.EventEmitter {
        // methods that emit and listen for custom events
        private initRaw(): void {
```

[10] https://nodejs.org/api/events.html#events_class_eventemitter

```
      // ...
      this.emit('checking-for-update');
    }

    public initialize(): void {
      this.on('checking-for-update', /* ... */ );
    }
}

export const Instance = new UpdateManager();
```

Since the access path representation does not reason about the inheritance hierarchy, the **new** `UpdateManager()` is represented abstractly as **require**(`events`).`EventEmitter`$_{\texttt{new}}$(). Other common custom exent usage patterns include extending the EventEmitter `prototype` [11] or including an EventEmitter as a class field [12]. In all these cases, the developers are encapsulating the base EventEmitter so as to build their own custom event-based infrastructure.

There are a variety of ways developers make use of custom event infrastructures in their code bases. Building something on top of the base EventEmitter is one of the most common, as we have discussed above; after this, the next most frequently observed emitter was objects of class `socket.io` Socket. Examining the documentation of the `socket.io` API about emitting events, we see that there are no standard events. Therefore, all events emitted on `socket.io` client or server based emitters are custom events. This corresponds to 8398 of the pairs (1911 client-side and 6487 server-side).

Manual analysis of a subset of remaining pairs. So far, we have determined that 14,866 or our 22,900 pairs mined correspond to emissions of custom events on either the base event emitter or via `socket.io`. This in turn does not mean that the rest of the pairs in our dataset do not correspond to custom events.

There are 2668 unique pairs remaining. Of these, we manually looked at a random sampling of 200, split across all the APIs[13]. In this manual analysis, we found that 82.5% correspond to custom events.

From this, we conclude that the vast majority of events manually emitted correspond to custom events. As discussed, the use of a custom event is local to its project and so patterns observed in other projects cannot be used to learn about its proper use. Therefore, our learning approach cannot be effectively used to identify bug patterns in lost events.

7 RELATED WORK

A considerable amount of research has focused on detecting and characterizing bugs in JavaScript applications, including: bug detection tools using static [9] and dynamic [6, 38] analysis, evaluations of the effectiveness of type systems for preventing bugs [21], development of benchmarks [26], and studies of real-world bugs [46].

The most closely related work to ours is by Madsen et al. [30]. They describe a static analysis for detecting dead listeners, lost events and other event-handling bugs based on the notion of an event-based call graph that augments a traditional call graph with edges corresponding to event-listener registration, event emission, and callback invocation. Event-handling bugs are detected by looking for patterns in these augmented call graphs. Unfortunately, their approach does not scale well because their context-sensitive analysis employs notions of contexts corresponding to the sets of events emitted and listeners registered, which may be exponential in the size of the program. This exponential behavior appears to manifest itself in practice, given that, on their largest subject program (which is a mere 390 LOC), one of their analyses incurs a running time of 17 seconds, and the other one does not terminate at all. Our approach targets

[11] Example: `kue`
[12] Example: `Azurite`
[13] All pairs manually analyzed are included in a table in the appendix.

only dead listeners, and only those cases where the event the listener is meant to handle is never emitted (excluding cases where it is emitted at a time when the listener is not registered). This allows us to use a simple and scalable static analysis in our mining phase, and rely on statistical reasoning over a large data set to offset the noise.

Our work also stands in a long line of research viewing bugs as “deviant behavior”: statistical methods are used to infer beliefs or rules that are implicit in the code, and violations of these rules are flagged as likely defects.

Engler et al. [16] distinguish between “MUST beliefs” and “MAY beliefs”. The former are directly implied by the code, and often boil down to simple data-flow properties: for example, dereferencing a pointer implies the belief that it is not a null pointer, and a subsequent null check of the same pointer is inconsistent with that belief. MAY beliefs, on the other hand, are patterns such as two functions that are often invoked in a particular order, which might reflect an implicit rule (such as the second one freeing a resoure allocated by the first one), or might be a coincidence. They use an analysis based on the z statistic to distinguish the two. Our work also aims to infer a MAY belief, but of a more complex kind than those considered by Engler, since the relationship between event emitters and events is many-to-many. Custom events pose an additional challenge that requires more sophisticated statistical methods than the z statistic.

The PR-Miner system [28] targets a broader class of rules: using frequent itemset mining, it extracts association rules $A \Rightarrow B$, where A and B are sets of program elements such as function calls. Such a rule expresses the observation that functions containing all elements in A also contain all elements in B, with a certain level of confidence. Violations of high-confidence rules are then likely to be bugs. Again, the relationship between event emitters and events does not immediately fit this pattern: association rules are “must” rules in the sense that if all elements in A are present then all elements in B must be present. In our case, however, the rules we want to infer are “may” rules in the sense that event emitters may emit one of a set of events, and vice versa.

WN-Miner [47] and PF-Miner [29] focus more narrowly on the problem of inferring temporal specifications, specifically pairs of functions f and g such that g must always be invoked after f, usually because it performs some sort of cleanup. Acharya et al. [3] generalize this to inferring partial orders between functions. Gruska et al. [25], on the other hand, generalize in a different direction and employ association rules of a similar kind as PR-Miner, but where the sets A and B now contain candidate function pairs, thus allowing inference of context-dependent specifications. Murali et al. [34] apply a Bayesian framework for learning probabilistic API specifications, which is more robust on noisy and heterogeneous data than more lightweight approaches. Although dead-listener detection shares some general principles with temporal-specification mining, the concrete setup is rather different and it is not immediately obvious that their techniques apply to our problem.

Monperrus et al. [33] propose *type usages* as a particularly useful kind of specification to infer for object-oriented programs: a type usage is a set of methods invoked on a variable of a given type, all within the body of a method with a given signature. They define a metric termed *s-score*, which can be used to identify type usages that are themselves rare, but similar to a very common type usage. This parallels our goal of finding rare event-emitter pairs where a pair with a different event or a different emitter is very common, although the technical details of our approach are again somewhat more complex to deal with the problem of custom events.

A significant amount of research has been devoted to the detection of *event races* using static [49] and dynamic [37, 40] analysis. Recent work has focused on event races that have observable effects [35], by classifying event races [48], and by developing specialized techniques focused on event races that occur during page initialization [5] or that are associated with AJAX requests [4]. The access paths used in this paper are not precise enough to capture the ordering constraints necessary for event-race detection, so our approach is not immediately applicable to this problem.

Other researchers have used statistical reasoning for predicting properties of programs for use in bug finding. Raychev et al. [41] derive probabilistic models from existing data using structured prediction with conditional random fields (CRFs). They apply their analysis to JavaScript programs to predict the names of identifiers and types of variables in new, unseen programs, and suggest that the computed results can be useful for de-obfuscation and adding or checking type annotations. Eberhardt et al. [15] apply unsupervised machine learning to a large corpus of Java and Python programs obtained from public repositories to infer aliasing specifications for popular APIs, which are then are used to enhance a may-alias analysis that is applied to applications using such APIs. The resulting enhanced analysis is demonstrated to lead to improvements in client analysis such as typestate analysis (by eliminating a false positive result) and taint analysis (by eliminating a false negative result). Chibotaru et al. [11] present a semi-supervised method for inferring taint analysis specifications. A propagation graph is inferred from each program in a dataset, and it is assumed that a small number of nodes corresponding to API functions is annotated as a source, sink, or sanitizer. To infer situations where unannotated nodes also play one of these roles, a set of linear constraints is derived from the propagation graph so that the solution to constraints represents the likelihood of unannotated nodes being a source, sink, or sanitizer.

Hanam et al. [27] present a technique for discovering JavaScript bug patterns by analyzing many bug-fix commits. They decompose commits into a set of language-construct changes, represent these as feature vectors, and apply unsupervised machine learning to identify bug patterns. The identified patterns are low-level issues such as dereferencing `undefined` and incorrect error handling. They do not discuss bug patterns related to event handling.

DeepBugs [39] aims to generate bug-fix changes automatically. By applying simple program transformations to code that is assumed to be correct, training data is obtained for a classifier that distinguishes correct from incorrect code. The approach is evaluated for three types of errors (swapped function arguments, wrong binary operator, wrong operand in binary operation), and detected dozens of real bugs, with a false positive rate of around 30%. It is unclear how well this approach would work for less syntactic bugs like the dead-listener bugs we consider.

Ryu et al. [42] present the SAFE tools for detecting type mismatch bugs that cause runtime errors (e.g., accesses to `undefined`) in JavaScript web applications. They construct simple models of browser runtime constructs such as the HTML Document Object Model (DOM) through a dynamic analysis; this is used as input for their bug detector. The SAFE tools differ from our work in three key ways: most importantly, the class of bugs SAFE tracks does not include dead-listener bugs; also, their target runtime is the browser while ours is Node.js; and, our analysis is purely static.

8 CONCLUSION

We have presented an approach for detecting dead listener patterns in event-driven JavaScript programs that relies on a combination of static analysis and statistical reasoning. The static analysis computes a set of listener-registration pairs $\langle a, e \rangle$ where a is an access path and e the name of an event, reflecting the fact that a listener is registered for e on an object represented by a. After applying the static analysis to a large corpus of JavaScript applications, statistical modeling is used to differentiate correct event listener registrations that are commonly observed from rarely observed cases that are likely to be incorrect. In a large-scale evaluation on 127,531 open-source JavaScript projects, our technique was able to detect 75 incorrect listener-registration pairs, while maintaining a precision of 90.9%.

We report on several additional experiments to better assess the impact of the data set analyzed by the statistical analysis, the utility of the results, and the practicality of the technique. One experiment revealed that the *selection* of the particular subset of data that statistical analysis is trained on does not substantially affect the choice of optimal configuration. On the other hand, we found the *size* of the subset used for training to have significant impact, with smaller training set sizes generally resulting in classifiers that have unstable precision and lower recall on the full data set. Furthermore, we demonstrated that our approach is effective at identifying incorrect listener registrations in real code bases: of the 30 issues we recently reported to developers of 25 open-source projects on GitHub, 7 were confirmed as bugs. While the statistical analysis requires a significant amount of compute time, we would expect this cost to be incurred infrequently, as APIs tend to evolve slowly. Checking a specific project for dead listeners typically takes no more than a few minutes for all but the largest projects.

As future work, we plan to explore the effectiveness of employing more precise notions of access paths, by using distinct representations for function calls where one or more of the arguments are string literals and one or more of the arguments is a function. In principle, this would enable us to distinguish access paths in the presence of nested event handlers.

ACKNOWLEDGMENT

The authors would like to thank Albert Ziegler for his insights and helpful discussions about the statistical modelling. E. Arteca and F. Tip were supported in part by the National Science Foundation grants CCF-1715153 and CCF-1907727. E. Arteca was also supported in part by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] 2020. eris bug fix commit. https://github.com/abalabahaha/eris/commit/6fcfc8c5472bdbc41f5cfbff499774534f6429ad.

[2] 2020. haraka bug fix commit. https://github.com/haraka/Haraka/commit/99cb9f93451e54b16cfad856de0c4811fb6d4ccf.

[3] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. 2007. Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, Ivica Crnkovic and Antonia Bertolino (Eds.). ACM, 25–34. https://doi.org/10.1145/1287624.1287630

[4] Christoffer Quist Adamsen, Anders Møller, Saba Alimadadi, and Frank Tip. 2018. Practical AJAX race detection for JavaScript web applications. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018.* 38–48. https://doi.org/10.1145/3236024.3236038

[5] Christoffer Quist Adamsen, Anders Møller, and Frank Tip. 2017. Practical Initialization Race Detection for JavaScript Web Applications. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* 66:1–66:22.

[6] Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. 2018. Finding broken promises in asynchronous JavaScript programs. (2018), 162:1–162:26. https://doi.org/10.1145/3276532

[7] Angular. 2020. https://angular.io/. Accessed: 2020-09-13.

[8] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy.* 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

[9] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. 2014. SAFEWAPI: web API misuse detector for web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014.* 507–517. https://doi.org/10.1145/2635868.2635916

[10] Joseph G Bryan and George P Wadsworth. 1960. Introduction to Probability and Random Variables. , 52 pages.

[11] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin T. Vechev. 2019. Scalable taint specification inference with big code. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[12] World Wide Web Consortium. 2020. UI Events - W3C Working Draft. https://www.w3.org/TR/DOM-Level-3-Events. Accessed: 2020-09-13.

[13] Pandas developers. 2020. pandas: Python Data Analysis Library. https://pandas.pydata.org/. Accessed: 2020-09-13.

[14] SciPy developers. 2020. SciPy. https://www.scipy.org/. Accessed: 2020-09-13.

[15] Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin T. Vechev. 2019. Unsupervised learning of API aliasing specifications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[16] Dawson R. Engler, David Yu Chen, and Andy Chou. 2001. Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP 2001, Chateau Lake Louise, Banff, Alberta, Canada, October 21-24, 2001*, Keith Marzullo and Mahadev Satyanarayanan (Eds.). ACM, 57–72. https://doi.org/10.1145/502034.502041

[17] Facebook. 2020. React: A JavaScript library for building user interfaces. https://reactjs.org. Accessed: 2020-09-13.

[18] OpenJS Foundation. 2020. EventEmitter Documentation. https://nodejs.org/api/events.html#events_emitter_on_eventname_listener. Accessed: 2020-09-10.

[19] OpenJS Foundation. 2020. jQuery. https://jquery.com/. Accessed: 2020-09-13.

[20] OpenJS Foundation. 2020. Node.js. https://nodejs.org. Accessed: 2020-09-13.

[21] Zheng Gao, Christian Bird, and Earl T. Barr. 2017. To type or not to type: quantifying detectable bugs in JavaScript. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 758–769. https://doi.org/10.1109/ICSE.2017.75

[22] GitHub. 2020. LGTM.com. https://lgtm.com/. Accessed: 2020-09-13.

[23] GitHub. 2020. QL standard libraries. https://github.com/Semmle/ql. Accessed: 2020-09-13.

[24] Electron Administrative Working Group. 2020. Electron: Build cross platform desktop apps with JavaScript, HTML, and CSS. https://electronjs.org. Accessed: 2020-09-13.

[25] Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. 2010. Learning from 6,000 Projects: Lightweight Cross-Project Anomaly Detection. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, Paolo Tonella and Alessandro Orso (Eds.). ACM, 119–130. https://doi.org/10.1145/1831708.1831723

[26] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. 2019. BugsJS: a Benchmark of JavaScript Bugs. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. 90–101. https://doi.org/10.1109/ICST.2019.00019

[27] Quinn Hanam, Fernando Santos De Mattos Brito, and Ali Mesbah. 2016. Discovering bug patterns in JavaScript. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 144–156. https://doi.org/10.1145/2950290.2950308

[28] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, Michel Wermelinger and Harald C. Gall (Eds.). ACM, 306–315. https://doi.org/10.1145/1081706.1081755

[29] Hu-Qiu Liu, Yu-Ping Wang, Jia-Ju Bai, and Shi-Min Hu. 2016. PF-Miner: A practical paired functions mining method for Android kernel in error paths. *J. Syst. Softw.* 121 (2016), 234–246. https://doi.org/10.1016/j.jss.2016.02.007

[30] Magnus Madsen, Frank Tip, and Ondrej Lhoták. 2015. Static analysis of event-driven Node.js JavaScript applications. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[31] Andreu Mas-Colell, Michael Dennis Whinston, Jerry R Green, et al. 1995. *Microeconomic theory*. Vol. 1. Oxford university press New York.

[32] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type Regression Testing to Detect Breaking Changes in Node.js Libraries. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands (LIPIcs)*, Todd D. Millstein (Ed.), Vol. 109. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 7:1–7:24.

[33] Martin Monperrus and Mira Mezini. 2013. Detecting Missing Method Calls As Violations of the Majority Rule. *ACM Trans. Softw. Eng. Methodol.* 22, 1 (2013), 7:1–7:25. https://doi.org/10.1145/2430536.2430541

[34] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Bayesian specification learning for finding API usage errors. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 151–162. https://doi.org/10.1145/3106237.3106284

[35] Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. 2015. Detecting JavaScript Races that Matter. In *Proc. 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 381–392.

[36] Forrest L Norvell. 2020. Add dynamic instrumentation to emitters. https://www.npmjs.com/package/emitter-listener/. Accessed: 2020-09-13.

[37] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. 2012. Race detection for web applications. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 251–262.

[38] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 314–324. https://doi.org/10.1109/ICSE.2015.51

[39] Michael Pradel and Koushik Sen. 2018. DeepBugs: a learning approach to name-based bug detection. (2018), 147:1–147:25. https://doi.org/10.1145/3276517

[40] Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective race detection for event-driven programs. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 151–166.

[41] Veselin Raychev, Martin T. Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 111–124. https://doi.org/10.1145/2676726.2677009

[42] Sukyoung Ryu, Jihyeok Park, and Joonyoung Park. 2018. Toward Analysis and Bug Finding in JavaScript Web Applications in the Wild. *IEEE Software* 36, 3 (2018), 74–82.

[43] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66. https://doi.org/10.1145/3188720

[44] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 598–608. https://doi.org/10.1109/ICSE.2015.76

[45] Benoit Vidis. 2020. Minimal Request Promise. https://github.com/benoitvidis/min-req-promise/. Accessed: 2020-09-13.

[46] Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. 2017. A comprehensive study on real world concurrency bugs in Node.js. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 520–531. https://doi.org/10.1109/ASE.2017.8115663

[47] Westley Weimer and George C. Necula. 2005. Mining Temporal Specifications for Error Detection. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science)*, Nicolas Halbwachs and Lenore D. Zuck (Eds.), Vol. 3440. Springer, 461–476. https://doi.org/10.1007/978-3-540-31980-1_30

[48] Lu Zhang and Chao Wang. 2017. RClassify: classifying race conditions in web applications via deterministic replay. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 278–288.

[49] Yunhui Zheng, Tao Bao, and Xiangyu Zhang. 2011. Statically locating web application bugs caused by asynchronous calls. In *Proceedings of the 20th international conference on World wide web*. ACM, 805–814.