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 used 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 (January 2021), 22 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 [8] defines the low-level event API supported by all modern browsers, while popular libraries such as jQuery [14], Angular [4] and React [12] 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 [19] 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 [15], which is dominant in server-side JavaScript, advocates an asynchronous programming style centred 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 [22]. 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. [22] 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* [23], 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.

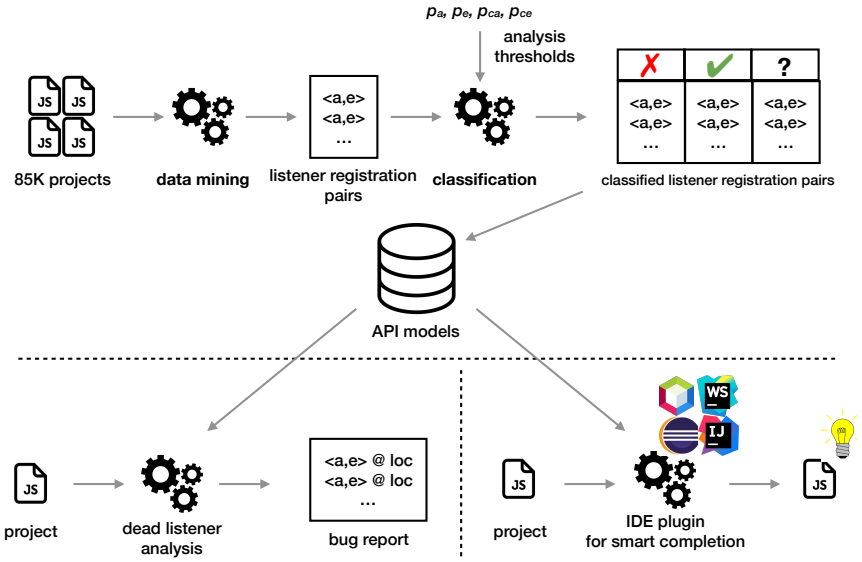

Fig. 1. Overview of approach.

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 [33, 34]. 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 cross-validation experiment. We partitioned our labeled set into two random halves, chose the optimal configuration for one half, and computed the true-positive rate of that configuration over the other half. We repeated this experiment 20 times. Our results show that while the optimal configuration for one half is not always optimal for the other half, it is still a very good choice.

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.

`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

```
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

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 [35].

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. [22].

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. [23]: 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:

a	::=	$\textbf{require}(m)$	an import of package m
	|	$a.f$	property f of an object represented by a
	|	$a()$	result of a function represented by a
	|	$a(i)$	ith argument of function represented by a
	|	$a_{\textbf{new}}()$	instance of a class represented by a

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 1627 instances of the first pair and 1654 of the second, but only two 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).

Discussion

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.

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:

```
1  const req = http.request(url);
2  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 path, but its type is `net.Socket`. This means that in some cases we cannot determine event registration correctness

[1]Note that the argument to the `request` method is not recorded in the access path.
[2]Note: argument indices start at 0, so the 1 in the access path for `res` indicates the *second* argument.

```
1  var eos = function(stream, opts, callback) {
2    // ...
3    if ( isRequest( stream)) {
4      stream.on('complete', /* ... */ );
5      stream.on('abort', /* ... */);
6    }
7    // ...
```

Fig. 3. Listener registration with type check

```
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

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:

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

While this does not affect the pair recorded for the first registration, the second becomes $\langle$**require**(`http`).`request`(1)(0).`on`(), `end`$\rangle$. 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*.

We did implement an alias analysis to recognize and remove the most common sources of access path alias, including chained listener registrations (such as the one shown here) and cyclic access paths; this is discussed in Section 5.1. We also performed an experiment to determine the effect this alias analysis has on result quality, and present these results in Section 6.

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.

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 look 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.

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

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).

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

```
1  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 [27] 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.

[4] Adapted from the `strongloop/strong-pm` project on GitHub

[5] Recall that in JavaScript the properties of an object are not fixed; properties can be added, overwritten, and deleted dynamically.

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`).$\texttt{ServerResponse}_{\textbf{new}}()$. 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: 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.

[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.

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 $\langle$**require**(`http`).`request`(1)(0), `timeout`$\rangle$ 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 cumulative distribution BCDF 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 $\langle a, \mathsf{doge}\rangle$ with an access path a that only occurs once and a rarity threshold p_a of 0.01, we get $\mathrm{BCDF}(1, 522, 0.01) \approx 0.03$, 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 [5] for identifying event registrations in JavaScript. Extensive libraries for writing static analyzers in QL are available [18], 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 [17], 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 [9], and the SciPy library [10] for the statistical computations.

5.1 Access Path Alias Removal

In Section 3, we discussed some sources of imprecision due to the construction of access paths. As part of this project, we extended the implementation of access paths available in QL 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* and *cyclic property assignments*. Our alias analysis in each of these cases is discussed below.

5.1.1 Chained listener registrations. A very common pattern in event-driven JavaScript is *chained listener registrations*. This refers to code such as (Figure 2 rewritten to use chained listener registrations):

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

Note how the listener for the `end` event is registered on the return value from the registration of the listener for the `data` event. And similarly, the listener for `timeout` is registered on the return value from the registration of the listener for `end`.

These listeners are all registered on the same object: `res`. However, as discussed in Section 3, the access paths for base of the listener registration are not the same. They are aliases.

This 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 [13].

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 from Section 3 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.

5.1.2 Cyclic property assignments. Another common source of access path aliases in our data was *cyclic property assignments*. This refers to cases where an object is assigned itself as a property or subproperty. This causes issues for the access path resolution: since the access path representation is recursive the presence of such cyclic assignments causes an explosion of access paths – every time an object containing this pattern is referenced, it has infinite aliases[7].

As a concrete example, consider the following code snippet, which is a condensed excerpt from the implementation of `readable-stream` [26].

```
1  var Stream = require('stream');
2  ...
3  module.exports = Stream.Readable;
4  module.exports.Stream = Stream;
5  ...
```

Here we see a cyclic property assignment: since `module.exports = Stream.Readable`, the assignment on line 4 is equivalent to `Stream.Readable.Stream = Stream`. This leads to infinite possible representations of `Stream`: `Stream.Readable.Stream`, `Stream.Readable.Stream.Readable.Stream`, etc.

We implement a check to detect and ignore this type of aliases. This check is for the presence of a property read that returns the same node (in the dataflow graph) as a property read on its base path.

In Section 6, we report on an experiment to determine the impact of this alias removal technique on the quality of our results.

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. Is there a threshold parameter configuration yielding an acceptable precision and recall rate?
RQ2. How sensitive is this choice of configuration to the training data set?
RQ3. Does our approach identify practically relevant mistakes?
RQ4. Is our approach practical in terms of performance/resources?

[7]The QL access path implementation caps the length at 10, and so our collected data is of course finite. However, this problem still causes a single object to have thousands of recorded aliases.

RQ5. What is the effect of our access path alias removal on the quality of our results?

We now address each of these questions in turn.

RQ1. 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 [33, 34]).

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 most likely 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.

In practice, the sheer number of pairs we are dealing with makes exhaustive manual classification impossible. Instead, we manually classified a set of pairs for those 18 packages for which our mining phase found the largest number of event registrations[8].

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 create 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, some access paths are imprecise. These imprecise paths paired with every event from the API form the list of *imprecise pairs*. We include the model for `http` in the appendix, as an example[9].

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 manually (or rather, semi-automatically) labeled pairs. This is the set we use to compute precision and recall rates for our model.

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

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

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

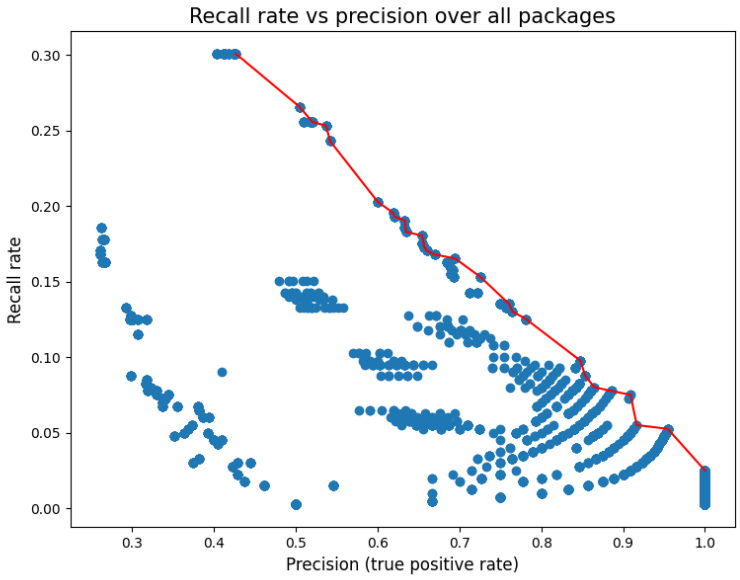

Fig. 5. Precision and recall for all configurations (blue dots); Pareto front in red

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*, 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 nine 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, the unique true-positive and false-positive counts, as well as 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 10%, 3%, 10%, and 5% respectively, results in a precision of 100% and recall of 6.8% over the labelled data set. This corresponds to 29 true positives and no false positivies; these true positive pairs occur 113 times in the mined data, across 47 projects.

To answer RQ1, then, we found that there are indeed configurations with more than 90% precision. The fifth 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 25% and 4%, and the confidence thresholds p_{ca} and p_{ce} are 10% and 1%, respectively; Over the labeled data set, this configuration yields 4 false positives and 42 true positives, for a precision of

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

Table 1. Optimal configurations with ≥80% recall

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

90.9%. The 42 true-positive pairs occur 75 times in total across 66 projects. All the false positives for this configuration were cases where the access path is overly imprecise.

RQ2. 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 more pairs, we perform a cross-validation experiment. In each round, we randomly halve our labelled data set into a training set and a validation set. Then we select the best configuration (i.e. highest recall with at least 90% precision) based on the training set, and compute its precision and recall on the validation set. Overall we perform 20 rounds of cross validation.

The results of the experiment are shown in Table 2. Each row represents one round. 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 $p_a = 0.25$, $p_e = 0.04$, $p_{ca} = 0.04$, $p_{ce} = 0.01$, which achieved a 92% precision with 10.7% recall, finding 23 true positive results. On the validation set, that same configuration resulted in a precision of 94.7% and recall of 8.4%, with 18 true positive results.

We see consistent results with the cross-validation experiment. Concretely: across the 20 rounds of the experiment, in the training set we see an average precision of 91.4% (standard deviation 1.0%) and an average recall of 10.6% (standard deviation 1.64%). Then, in the validation set we see an average precision of 86.9% (standard deviation 7.8%) and an

```
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`

average recall of 10.2% (standard deviation 1.6%). 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 $p_a = 0.25$ is found in 19 runs, $p_e = 0.04$ in 16 runs, $p_{ca} = 0.1$ in 7 runs, and $p_{ce} = 0.01$ in 5 runs. In conclusion, the choice of training set does not substantially affect the choice of optimal configuration.

RQ3. To qualitatively assess the usefulness of our approach, we performed 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 [10], one where the developers did not remember what the code was supposed to do, and one that was a false positive (due to manually emitting the event elsewhere in the project). We did not yet receive a response for the remaining 21 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 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.

RQ4. 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

[10] Two have been addressed, links to commits: `eris` bug; and `Haraka` bug

With alias removal		Without alias removal	
% Precision	% Recall	% Precision	% Recall
100.0	3.0	100.0	2.4
95.8	5.8	95.4	5.1
92.3	6.0	91.7	5.4
90.9	7.5	90.9	7.3
88.6	7.8	88.6	7.6
86.5	8.0	86.5	7.8
85.4	8.8	85.4	8.5
84.8	9.8	84.8	9.5

Table 3. Pareto front comparison, for data with and without access path alias removal

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.

RQ5. With this last research question, we determine the effect of our access path alias removal on the quality of our results. To do this, we ran the data mining and classification using a version of the analysis *without* the alias removal. Then, we computed the Pareto front for this data, again maximizing for both precision and recall[11].

Table 3 shows a side-by-side comparison of the Pareto fronts with and without alias removal, for recall ≥80%. Note that the Pareto front with alias removal corresponds to the first two results columns in Table 1. For example, the first row reads as follows: when the analysis is run with alias removal, the point on the Pareto front with 100% precision has 6.8% recall; when the analysis is run *without* the alias removal, the point on the Pareto front with 100% precision has 3.6% recall.

We see that for the same precision, the addition of our alias removal technique results in a consistent doubling of the recall rate. From this, we conclude that the alias removal doubles the quality of our results. This is a significant improvement, and we consider it well worth the slight increase in the complexity of the analysis.

The full Pareto front for the data without alias removal (i.e., we regenerated Table 1 with this data) is included in the appendix.

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

[11] We computed the equivalent of Table 1 for this data – this is included in the appendix.

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 true-positive rates do 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%).

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.

7 RELATED WORK

A considerable amount of research has focused on detecting and characterizing bugs in JavaScript applications, including: bug detection tools using static [6] and dynamic [3, 29] analysis, evaluations of the effectiveness of type systems for preventing bugs [16], development of benchmarks [20], and studies of real-world bugs [36].

The most closely related work to ours is by Madsen et al. [22]. 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. Our approach targets 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.

A significant amount of research has been devoted to the detection of *event races* using static [38] and dynamic [28, 31] analysis. Recent work has focused on event races that have observable effects [25], by classifying event races [37], and by developing specialized techniques focused on event races that occur during page initialization [2] or that are associated with AJAX requests [1]. 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. Raychev et al. [32] 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. [11] 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. [7] 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.

Murali et al. [24] focus on detecting API usage errors in Android apps using a Bayesian framework for learning probabilistic specifications and using the inferred specifications to detect likely bugs. An instantiation of the framework is discussed that associates probabilities with API call sequences and flags sequences that seem anomalous. Our approach also relies on statistical reasoning rather than manually constructed/inferred semantic models but we use a lightweight statistical model without involving machine learning.

Hanam et al. [21] 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 [30] 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. [**?**] 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 some 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 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.

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] 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

[2] 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.

[3] Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. 2018. Finding broken promises in asynchronous JavaScript programs. *PACMPL* 2, OOPSLA (2018), 162:1–162:26. https://doi.org/10.1145/3276532

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

[5] 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

[6] 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

[7] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin T. Vechev. 2019. Scalable taint specification inference with big code. In *PLDI*.

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

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

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

[11] Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin T. Vechev. 2019. Unsupervised learning of API aliasing specifications. In *PLDI*.

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

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

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

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

[16] 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

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

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

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

[20] 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

[21] 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

[22] Magnus Madsen, Frank Tip, and Ondrej Lhoták. 2015. Static analysis of event-driven Node.js JavaScript applications. In *OOPSLA*.

[23] 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.

[24] 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

[25] 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.

[26] Node.js. 2020. Readable Stream implementation. https://github.com/nodejs/readable-stream/blob/master/readable.js. Accessed: 2020-09-13.

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

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

[29] 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

[30] Michael Pradel and Koushik Sen. 2018. DeepBugs: a learning approach to name-based bug detection. *PACMPL* 2, OOPSLA (2018), 147:1–147:25. https://doi.org/10.1145/3276517

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

[32] 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

[33] 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

[34] 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

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

[36] 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

[37] 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.

[38] 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.