AsyncLambdaTester: Automatically Testing JavaScript APIs with Asynchronous Callbacks

ANONYMOUS AUTHOR(S)

Test generation is an effective means for automatically testing libraries. More and more libraries provide APIs that accept callbacks and then invoke them asynchronously, to avoid blocking the main computation, e.g., when accessing some kind of resource. Automatically testing such APIs is challenging because the test generator must not only call an API function but also generate a function to be passed as a callback. Moreover, an effective test must combine multiple calls to related API functions, e.g., when a parameter available in a callback given to one function should be used as an argument given to another function. To test such dependencies, simply sequencing calls, as done in existing test generators, is insufficient. This paper presents AsyncLambdaTester, the first test generator aimed at APIs with asynchronous callbacks. The approach addresses the above challenges by both sequencing and nesting API calls. Nesting here means that when a callback is invoked asynchronously, it triggers more API calls, which can use values available only once the callback has been invoked. Our evaluation applies the idea to 10 popular JavaScript libraries that contain 132 API functions with asynchronous callbacks. The approach achieves high statement coverage (between 32% and 87%) and detects various behavioral differences in a regression testing scenario (on average, 1.3 per 100 generated tests), some of which are bugs. Compared to a state-of-the-art tool, AsyncLambdaTester exercises otherwise missed behavior and reaches interesting behavior faster.

Additional Key Words and Phrases: asynchronous programming, test generation, JavaScript, testing

ACM Reference Format:
Anonymous Author(s). 2018. AsyncLambdaTester: Automatically Testing JavaScript APIs with Asynchronous Callbacks. *J. ACM* 37, 4, Article 111 (August 2018), 24 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Test generation is an important technique to automatically test libraries by creating unit-level tests. Most test generators create tests that contain a sequence of calls to the tested API functions. In such a sequence, a call may use the return value of any previous call as an argument, enabling the testing of dependent API functions. Different test generators take different approaches for selecting which functions to call and which arguments to pass into them, e.g., random inputs [Ciupa et al. 2008; Csallner and Smaragdakis 2004], feedback from executions [Fraser and Arcuri 2011; Pacheco et al. 2008, 2007], and symbolic reasoning [Sen et al. 2005; Visser et al. 2004; Xie et al. 2005].

Existing work on test generation ignores a class of APIs that is becoming more and more prevalent: Higher-order functions with asynchronous callbacks, i.e., functions that accept another function as an argument and then invoke that other function asynchronously. The key benefit of asynchronous callbacks is that invoking the callback does not block the main computation, which is useful, e.g., when accessing some kind of resource or when triggering a long-running computation. Asynchronous callbacks have been shown to be popular [Gallaba et al. 2015], but also prone to mistakes and surprising behavior [Okur et al. 2014; Wang et al. 2017].

0004-5411/2018/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

As a motivating example, consider a hypothetical library for accessing the file system in JavaScript applications. Due to its single-threaded execution model and event-oriented programming style, asynchronous callbacks are particularly prevalent in JavaScript [Gallaba et al. 2015]. Suppose the library offers API functions for creating, writing, and reading files, directories, symbolic links, etc. To allow clients to chain multiple calls to the library, the API functions each take a callback argument, which gets invoked once the file system operation has finished. For example, client code to first create a file and to then write some data into it may look like this:

```
1  fsLib.createFile("foo.txt", function callback(err, fileHandle) {
2    fsLib.writeFile(fileHandle, data);
3  });
```

To test whether the library behaves as expected under various usage scenarios, test generators can automatically create client code. Unfortunately, current test generators are not effective at testing asynchronous APIs, such as the above example. One problem is that most test generators [Fraser and Arcuri 2011; Pacheco et al. 2007] do not consider callbacks at all, i.e., they will never pass a function as an argument into another function. One notable exception is LambdaTester [Selakovic et al. 2018], a feedback-directed test generator specifically designed to exercise APIs that accept callback functions. However, LambdaTester targets only synchronously invoked callbacks, and cannot effectively test functions that invoke their callbacks asynchronously. The reason is twofold. First, LambdaTester fails to identify API functions that receive an asynchronously invoked callback argument, and hence, never passes callbacks into such functions. Second, the generated tests arrange calls to API functions in a sequence, but never nest one API call into another, as in the above client code. As a result, the state of the art approach cannot effectively test multiple API functions that depend on each other.

This paper presents AsyncLambdaTester, the first test generator aimed at libraries with asynchronous APIs. At the core of the approach is a novel tree-based representation of test cases, which allows for growing a test case by either sequencing API calls, i.e., adding sibling nodes, or by nesting API calls, i.e., adding child nodes. Sequencing here means that one API function is called after another, similar to previous test generators [Csallner and Smaragdakis 2004; Fraser and Arcuri 2011; Pacheco et al. 2007; Selakovic et al. 2018]. Nesting here means that in the function body of a callback passed to one API function, another API function is called, a feature not supported by prior work. The test generator iteratively extends test cases in a feedback-directed way. Our implementation targets JavaScript, where asynchronous callbacks are particularly prevalent and where generating effective tests is particularly challenging due to the absence of statically declared types.

In a nutshell, the AsyncLambdaTester approach consists of two broad steps. First, an API discovery step determines which API functions accept callbacks and whether these callbacks are invoked synchronously or asynchronously. Second, a feedback-directed test generation step grows test cases by sequencing and nesting API calls. To address the question of which API functions to combine with each other, the approach can optionally be guided by nesting examples mined from existing API clients, which yields more realistic and effective tests.

We evaluate AsyncLambdaTester with ten popular JavaScript libraries that include a total of 356 API functions, 142 of which expect callbacks. The approach effectively covers substantial parts of the tested libraries, with 32% to 87% statement coverage, depending on the library. We also apply AsyncLambdaTester in a regression testing scenario, where it finds 281 behavioral differences between code in consecutive commits, including bug fixes, breaking API changes, and newly introduced bugs. An empirical comparison with the best existing technique for testing APIs with callbacks [Selakovic et al. 2018] shows that our approach reaches and exceeds the coverage

achieved by the state-of-the-art in a fraction of the testing budget, and that it finds otherwise missed behavioral differences.

In summary, this paper contributes the following:

- The first test generation approach aimed at functions with asynchronous callbacks.
- An API discovery mechanism that automatically detects the majority of callback-expecting API functions in a library, without any specification or other information about the library.
- A tree-based representation of test cases that captures sequencing and nesting of API calls.
- A feedback-directed algorithm for incrementally growing tree-shaped tests.
- Empirical evidence that the approach is effective at exercising JavaScript APIs with asynchronous callbacks, and that it outperforms the current state-of-the-art.

2 BACKGROUND AND MOTIVATION

2.1 Background: (A)synchronous Callbacks and How to Test Them

This paper presents a test generation technique for testing higher-order functions that invoke their callbacks synchronously or asynchronously. A function is a *higher-order function* if it expects another function to be passed as an argument. Moreover, if a function f receives a callback function *cb* as an argument, then we will say that f *invokes cb synchronously* if f directly invokes *cb*, or if f calls another function that invokes *cb* synchronously. Furthermore, we will say that f *invokes cb asynchronously* if the execution of f causes *cb* to be invoked from the main event loop at some later time.

Our work builds on previous research on feedback-directed random test generation [Pacheco et al. 2007], in which tests are generated in an iterative manner. In each iteration, a previously generated test is extended with a call to a function with arguments consisting of randomly chosen values and/or values computed by previously called functions. The test is then executed. If it terminates normally, it is a candidate for being extended further in a subsequent iteration.

The presence of higher-order functions in the software under test introduces several new challenges. First, it requires functions to be passed as arguments, so a mechanism is required for synthesizing function values. Previous work [Selakovic et al. 2018] explores several approaches for synthesizing function values, including functions that return a random value. However, their work focused only on higher-order functions that invoke their callbacks synchronously.

Our research extends the concept of feedback-directed test generation to higher-order functions that invoke their callbacks asynchronously. The presence of asynchronous callbacks causes some complications for testing. Most significantly, the order in which asynchronous callbacks are invoked may be non-deterministic. Consider a situation where an API declares the following two methods:

```
4  function f(x, cb1){ ... } /* invokes callback cb1 asynchronously */
5  function g(x, cb2){ ... } /* invokes callback cb2 asynchronously */
```

and assume that a test invokes these functions as follows:

```
6  function test1(){
7    f("foo", function zip(){ ... }
8    g("bar", function zap(){ ... }
9  }
```

Then, in general, the order in which the functions `zip` and `zap` are invoked may vary non-deterministically, causing the test to potentially produce different results depending on the order in which event handlers are scheduled. The only way to guarantee that asynchronously invoked callbacks are executed in a specific order is to place method calls inside callbacks. For example, the following test:

```
16  // Omitted for clarity:
17  //  * Try-catch around each call to an API function.
18  //  * Print statements to log arguments and return values.
19  //  * Print statements to log control flow.
20
21  let fs_extra = require("fs-extra");
22
23  var arg590 = "a/b/test";
24  var arg593 = null;
25
26  let r_126_0 = fs_extra.ensureFile(arg590, function callback(a, b, c, d, e) {
27      let r_ 126_0_0 = fs_extra.readJson(arg590);
28      return false;
29  });
30
31  let r_126_1 = fs_extra.stat(arg593);
```

Fig. 1. Simplified example of a generated test for the API functions of the `fs-extra` library.

```
10    function test2(){
11      f("foo", function zip(){
12        g("bar", function zap(){ ... }
13        ...
14      }
15    }
```

will execute `zap` after `zip` (if `zap` executes at all, which is not guaranteed). The latter form is commonly used in asynchronous programming in cases where a specific order is desired, e.g., because one call depends on a state established by another.

2.2 Motivating Example

As a real-world example, we will consider the `fs-extra` library[1], which provides functionality for accessing, reading, and writing the file system beyond the built-in functionality provided by Node.js. In particular, the package defines functions `ensureFile`, which creates a file in case it is missing, `readJson`, which reads JSON data from a file, and `stat` for checking the status of a file. Each of these functions has a number of parameters, some of which are optional, and some of which are expected to be synchronously or asynchronously invoked callback functions.

Figure 1 shows a generated test for several API functions of the `fs-extra` library.[2] The test includes three calls to API functions that are composed using a combination of sequencing and nesting. After initializing some variables to store the argument values used in the calls (lines 23–24), the test invokes `ensureFile` (lines 26–29) with two arguments, the second of which is a callback function. This call is followed by a call to `stat` (line 31). Inside the body of the callback passed to `ensureFile`, the test contains another call of an API function, which invokes `readJson` (line 27). The first argument given to `ensureFile` is the same as the first argument given to `readJson`, i.e., the test first creates a file in case it is missing and then tries to read JSON data from this file.

2.3 Challenges

Creating test cases such as the example in Figure 1 involves several challenges not addressed by previous work.

[1] See https://www.npmjs.com/package/fs-extra.
[2] The test is slightly simplified for the sake of illustration. The full versions of generated tests are included in the supplemental material.

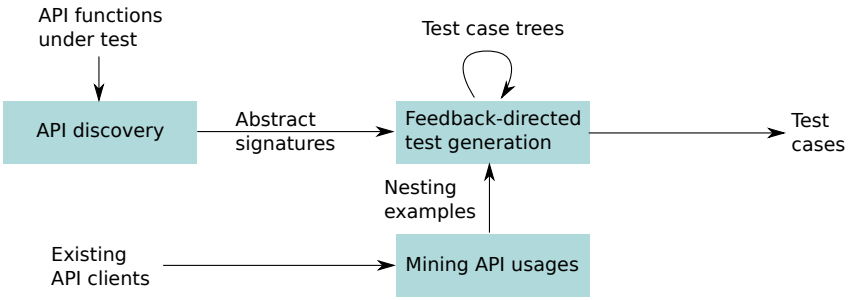

Fig. 2. Overview of the approach.

(1) One challenge is how to find out which APIs expect (a)synchronous callbacks and at what argument positions these callbacks should be passed. In the example, the second argument of `ensureFile` is expected to be a callback, which gets invoked asynchronously. Since JavaScript is dynamically typed and most libraries come without any formal specification of the signatures of their APIs, our approach needs to infer the signatures of functions as a prerequisite to generating effective tests.
(2) The second challenge is about how to compose multiple API calls into a test. While existing work focuses on sequencing calls, i.e., one call statement after another, sequencing alone is insufficient for testing asynchronous higher-order functions. The reason is that asynchronous callbacks often are invoked after a particular event has been triggered or after a particular state has been established, which may be the appropriate time to invoke another API function. In our example, the call to `ensureFile` creates a previously missing file, and only once this file has been created, one can test the behavior of `readJson` on an existing file. To this end, the test nests the call to `readJson` inside the callback given to `ensureFile`, which guarantees the `readJson` call to be invoked after `ensureFile` has performed its duty. If, instead of nesting the `readJson` call into the callback, the test would put both calls in sequence, the test would attempt to read from a non-existing file.
(3) The third challenge relates to the second, and it is about how to compose API calls in a realistic way. Without any prior knowledge, the chances of nesting a call to `readJson` into the callback given to `ensureFile`, both with the same first argument, are rather small. To increase the chances that our approach nests API calls in ways that represent realistic API usages, it requires some knowledge of typical API usages. Because our approach aims at fully automatic test generation, the challenge is to obtain and use this knowledge in an automatic way.

3 OVERVIEW

This section presents a high-level overview our approach for automatically testing JavaScript APIs with asynchronous callbacks, which is described visually in Figure 2. Our approach consists of three main components, which address the three challenges described in Section 2.3.

Given a set of API functions to test, the first step is an *automated API discovery*, which probes the API functions under test to determine if and where they expect callback arguments. Then, the *abstract signatures* discovered in this phase serve as input to a *feedback-directed test generation algorithm*, which is the core of the approach. The test generation is centered around a tree-based representation of test cases, called *test case trees*, which the approach iteratively grows into full test cases. To inform decisions about how API functions should be combined, the approach also *mines*

API usage patterns in existing open-source clients of the libraries under test. This information is passed to the test generator and used to support nesting decisions. The end product of this process is a set of test cases, which can be used in a variety of applications, e.g., regression testing. In the next sections, we delve into these steps in more detail.

4 API DISCOVERY

The first step in test generation is determining what to test: Given a library under test as input, AsyncLambdaTester retrieves the set of all function properties offered by the library object. However, as JavaScript is a dynamically typed language, AsyncLambdaTester initially does not know anything about these functions beyond their names. The API discovery step of the approach addresses this problem by probing the functions to determine at what parameter positions the function expects callback arguments and whether these callbacks are invoked synchronously or asynchronously. This information is recorded as a set of *abstract signatures*.

Definition 1 (Abstract signature)**.** An abstract signature for a function f is a tuple $(arg_1, ..., arg_n)$, where each arg_i is one of the following three kinds of elements:

- *async* to express that an argument is an asynchronously invoked callback,
- *sync* to express that an argument is a synchronously invoked callback, or
- the _ symbol to express any non-callback argument.

A single function may have zero, one, or multiple signatures[3]. Zero signatures indicates that the discovery phase failed to find any signatures for the function, in which case the approach falls back to a default test generation strategy that does not pass any callbacks. Multiple signatures are inferred in cases where functions are overloaded. For example, the `outputJson` function from `fs-extra` has two signatures: (_, *async*) and (_, _, *async*). The reason is that `outputJson` has an optional second argument, and always takes a callback function as the last argument.[4] In our experience, this type of overloading is extremely common because in JavaScript APIs, there are often some optional arguments preceding the (final) callback argument.

To discover signatures for a given API function under test, AsyncLambdaTester repeatedly invokes the function with randomly generated arguments. The function is tested multiple times with a callback at each argument position and with no callback arguments, and with different numbers of arguments. During the execution of these tests, AsyncLambdaTester tracks if and when callbacks are invoked and whether the function terminates successfully or throws an error. Then, using this information, it infers the corresponding signatures.

Algorithm 1 summarizes this API discovery phase. For each function f in the API, the algorithm generates a number of tests corresponding to a specified test budget $nTests$. Each of these tests calls f with a randomly generated argument list. The flag variable $testingCB$ indicates whether a callback should be included in the arguments or not; it is negated at every iteration of the while loop so the algorithm alternates between generating tests with and without a callback argument. This test is then executed and the feedback is examined to determine if this configuration of arguments corresponds to a valid signature for f; if so, this signature is added to the return list. If at the end of $nTests$ no valid signatures have been discovered for f, then we add the default "callbackless" signature to represent this failure of discovery. We test only a single callback argument at a time, i.e., our approach will not discover signatures with multiple callback arguments. The rationale is that API functions with multiple callbacks are relatively rare. Extending the algorithm

[3]For concision, we use "signature" to refer to "abstract signature" henceforth.
[4]https://github.com/jprichardson/node-fs-extra/blob/HEAD/docs/outputJson.md

Algorithm 1 Discovery of API signatures

Input: Input: Set F of API functions, test budget $nTests$, range of arguments to test with $nArgs$
Output: Output: Set S of discovered signatures

1: $S \leftarrow \emptyset$
2: **for** function f in F **do**
3: $validSigFound \leftarrow False$ // flag to indicate a valid signature was found for f
4: $cbPos \leftarrow 0$ // position of callback argument
5: $testingCB \leftarrow True$
6: $i \leftarrow 0$
7: **while** $i < nTests$ **do**
8: // generate test: with callback at arg. position cbPos if testing callbacks
9: $T \leftarrow generateTest(cbPos, testingCB)$
10: status $\leftarrow execute(T)$
11: **if** status == "async" *or* status == "sync" *or* status == "correct_cb_less" **then**
12: // generate signature with _ for all non-callback positions
13: // (if there is a callback arg., this position is marked by "status")
14: $sig \leftarrow generateSignature(cbPos, \text{status})$
15: $S.add(sig)$
16: $validSigFound \leftarrow True$
17: **if** $testingCB$ **then**
18: // use cbPos to ensure we try callbacks at all argument positions;
19: // cycle through them all again once we've tried them all
20: $cbPos \leftarrow (cbPos + 1)\ \%\ nArgs$
21: $testingCB = \neg\ testingCB$
22: $i \leftarrow i + 1$
23: **if** not $validSigFound$ **then** $S.append(\{f, -1, \text{"default\_cb\_less"}\})$
return S

to support multiple callbacks is straightforward but increases the computational complexity of the API discovery.

Concretely, the generated tests for a function `api_function` are structured as follows:

```
32  let callback = function() { console.log("Callback executed"); }
33  try {
34      // try calling the specified API function
35      api_function( ..., callback, ...);
36      console.log("API call executed");
37  } catch(e) {
38      console.log("Error in API call");
39  }
40  console.log("Test executed");
```

The test contains several print statements inserted before, in, and after the callback to track if and when the callback gets invoked. There is also a print statement after the API call, which will execute if the call terminates without error, and another print statement in the **catch** clause, which will execute if there is an error. Based on this information, lines 11 to 16 of the algorithm construct a signature and add it to the set of all signatures for the current function.

To distinguish between callbacks invoked synchronously and asynchronously, and between calls that fail and succeed, the algorithm distinguishes between several kinds of test output:

- If `"Callback executed"` comes *before* `"Test executed"`, this indicates the callback is executed *synchronously*.
- If `"Callback executed"` comes *after* `"Test executed"`, this indicates the callback is executed *asynchronously*.
- If `"Callback executed"` is never printed, this means that either this argument position does not correspond to a callback, or that there was an error in invoking `api_function`.
  - If `"Error in API call"` is printed, then there was an error in invoking `api_function`.
  - If `"API call executed"` is printed, then the API call was successful.

If the callback is executed at all, then the algorithm adds this argument configuration to the list of discovered signatures. If the callback is not executed, or there was no callback argument in the generated test, but there was no error in the API call, then we add this information to the list of discovered signatures. However, if no callback was executed and there was an error in the API call, we cannot be sure what caused the error (i.e., whether the argument being tested was incorrect, or if another of the randomly generated arguments lead to an error); in this case, the algorithm continues to explore the signatures. If the end of the test budget is reached and no valid signatures have been found, then we add an entry that indicates this function has no valid signatures and is "callback-less" by default to the list of discovered signatures, as we see on line 23 of Algorithm **??**.

Our approach for discovering signatures relates to a similar step in the existing LambdaTester [Selakovic et al. 2018], but differs in several key ways. First, our algorithm not only discovers synchronously invoked callbacks, but also track asynchronously invoked callbacks, and distinguishes both kinds from each other. Second, our algorithm also discovers signatures that take *no* callback arguments. These differences allow AsyncLambdaTester to automatically test a larger set of behaviors, which contributes to a significant increase in testing effectiveness (Section 6).

The output of the API discovery is the set of discovered signatures for each function offered by the library under test. When generating tests, the number of arguments and callback positions (if the function has a callback argument) are informed by these discovered signatures.

5 FEEDBACK-DIRECTED GENERATION OF TEST CASE TREES

Our algorithm for generating tests is based on a tree-shaped representation of test cases called a *test case tree*. The following motivates and describes test case trees and then presents the core test generation algorithm.

5.1 Test Case Trees

Effectively testing API functions that receive asynchronous callbacks requires to not only *sequence* API calls, as done in previous test generators, but to also *nest* API calls into the bodies of callback functions. To represent test cases that support both sequencing and nesting, we present a novel intermediate representation of test cases:

Definition 2 (Test case tree)**.** Let V be a map of variable names to non-callback values, called value pool, and S be a map of function names to their abstract signatures. Then, a *test case tree* is an ordered tree with two kinds of nodes and two kinds of edges. Nodes are either

- a *call node* of the form $r = api(a_1, ..., a_k)$, meaning that function api is given arguments $(a_1, ..., a_k)$ and yields return value r. If $s_i \in \{sync, async\}$ for some signature $(api, (s_1, ..., s_n)) \in S$, then a_i may be a callback function, whereas $a_i \in V$ or it is a return value in scope otherwise.
- a *callback node* of the form $cb(p_1, ..., p_k)$, meaning that callback function cb receives parameters $(p_1, ..., p_k)$.

Edges are either

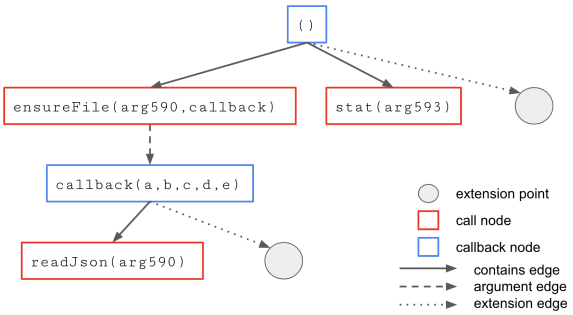

Fig. 3. Test case tree representation of the test in Figure 1, with extension points annotated.

- a *contains edge* that goes from a callback node to a call node, meaning that there is a call in the body of the callback function, or
- an *argument edge* that goes from a call node to a callback node, meaning that a call is given a callback function as an argument.

The root node of a test case tree is a special callback node that corresponds to the function that contains the entire test case. The children of any node in a test case tree are ordered. The order of the call nodes under a callback node represents the sequential order in which calls occur in the body of a callback function.

For example, consider the test case in Figure 1 and its corresponding tree representation in Figure 3. Each API call in the test case is represented by a call node. The nodes of calls that receive callback arguments each have a corresponding callback node as a child. For example, the callback function given to `ensureFile` at line 26 is represented by the callback node `callback(a, b, c, d, e)`. Calls nested within the body of a callback function are represented as children of callback nodes. For example, the call to `readJson` on line 27 corresponds to the lower-left call node of the tree. The value pool of the test consists of two entries, which map the variable names to their respectively assigned values in lines 23 to 24.

A call in a test case can use as arguments only values that are available at the call site according to the scoping rules of JavaScript. We say that a test case is *well formed* if, for all calls, its arguments are bound to some declaration when the call is reached during the execution of the test case. Given our tree representation of a test case, a test case is well-formed if and only if the following holds:

Definition 3 (Well-formedness)**.** In a well-formed test case tree, each argument of a call node n is one of the following:

- A random primitive value, inserted by referring to one of the entries in the value pool V.
- A random object value (array literal, object literal, or function), also inserted by referring to one of the entries in V.
- The return value r of a call node n' that is a left sibling of n or a left sibling of any ancestor call node of n (these represent return values of calls executed before reaching the call in n).
- A parameter p_i of a callback node on the path between the root node and n (these represent formal parameters of a surrounding callback function).
- A callback function cb_x, where n has a callback node that represents cb_x as a child.

For illustration, consider again the example in Figure 3, which shows a test case tree that is all arguments are well-defined. E.g., in the call of `ensureFile`, the first argument is `arg590`, which is

a randomly generated primitive value in the value pool (a string literal, defined on line 23), and the second argument is a callback node. In contrast, the call of `stat` could not use, e.g., `a` as an argument because `a` is not in the parameter list of any callback node on the path between `stat`'s node and the root node.

5.2 Test Generation Algorithm

Our test generation algorithm creates test cases iteratively, by repeatedly extending an existing test case with another call. The algorithm is based on the notion of *extension points*, which represent locations in a given test case where a new call node could be inserted. Given a test case tree, there is an extension point for each callback node that is executed during test execution, which adds another child node to the already existing child nodes, at the right-most position. In particular, given a callback node with no children, there is an extension point for adding a first child node. The test tree representation in Figure 3 has been annotated to illustrate the extension points of the current test case tree, and the contains edges that would be added if the test was extended at these points (labeled the *extension edges*). In the source code representation of the tree, the extension points correspond to adding a call right after lines 27 and 31.

Algorithm 2 summarizes our feedback-directed approach for creating test cases. The algorithm starts with an empty test case tree, which consists only of the root node. The main loop extends the test with a random number of calls (between 1 and 4) per iteration. This algorithm depends on several helper functions that choose from a set of options; these are described below.

Choosing an extension point. Instead of adding calls at arbitrary extension points, the algorithm executes each test after construction and determines which extension points are reached during the test execution. Those extension points that *are* reached during test execution are added to a list of *active extension points*. Then, in future iterations of the test generation algorithm, these extension points are potential starting points for new tests. The process of selecting one such point is done by calling the helper function *chooseExtensionPoint*, which simply returns a random element from the list of previously identified extension points. Note that if an extension point is used once, it is not removed. That is, multiple tests might start from the same extension point, and the set of extension points grows monotonically during the test generation process.

There are two main reasons for *not* reaching an extension point: exceptions thrown by the tested APIs and callback functions that are never invoked. By examining feedback from test executions, the algorithm avoids creating future tests that build on code that will never execute. The algorithm is called repeatedly, each time starting from a randomly selected extension point, from the pool of previously identified extension points.

Choosing a function to call. Given a specific extension point, the next decision is what function to call, which in Algorithm 2 is performed by *chooseFunction*. AsyncLambdaTester balances two requirements. On the one hand, the generated tests should cover as many of the given functions as possible. On the other hand, we do not want to prescribe a specific order in which the functions are selected. To this end, the approach assigns to each of the given functions a weight and then takes a weighted, random decision. Initially, all functions have uniformly distributed weights. When a function is selected by *chooseFunction*, the current weight is divided by a constant factor (four in our current implementation). Note that this reduction is done every time a function is chosen, i.e., if the same function is chosen twice, its weight is divided by the constant factor twice. In addition to the above, *chooseFunction* is guided by a mined model of nested API functions, as explained in detail in Section 5.3.

Algorithm 2 Feedback-directed test generation.

Input: Set F of API functions, number $nTests$ of tests to generate, mined nesting examples M, map S of function names in F to their discovered signatures, value pool V
Output: Set of tests T

```
1:  T ← [] // start with empty list of tests
2:  // start with empty extension point: root node, no function name, and value pool
3:  E ← [ {(), _, V} ]
4:  while T.length ≤ nTests do
5:      curExt ← chooseExtensionPoint(E) // choose randomly from active extension points
6:      A ← available args at curExt
7:      numCallsToAdd ← choose random value ∈ [1..4]
8:      // current test starts as the extension point
9:      curTest ← copy a test with extension point curExt
10:     allNewArgs ← []
11:     for i from 0 to numCallsToAdd do
12:         f ← chooseFunction(F, cur_ext, M)
13:         args ← chooseArguments(f, M, S, A)
14:         // extend current test
15:         curTest ← extendTest(curExt, f, args)
16:         curExt ← either right sibling of f or nested in a callback to f
17:         // add arguments to list of available
18:         allNewArgs ← allNewArgs + args+ return of f
19:     feedback ← execute(curTest)
20:     // if test is successful, add new returns and args to potential args of future
21:     // calls that start from this extension point
22:     // then, add the newly found extension point to the list
23:     if feedback == SUCCESS then
24:         A ← A + allNewArgs
25:         E ← E + [{curExt, f, A}]
26:     T ← T + [t]
27: return T
```

Choosing arguments to pass into a function. Once a function has been selected for testing, arguments need to be generated for it. This is done by consulting the list of discovered signatures, i.e., the output of the discovery phase. If there are multiple signatures for the function, one is chosen randomly from the list. The signatures inform the test generation of the number of arguments that the function should be passed, and which (if any) are callback arguments. If there are no signatures for the function, then a random number of arguments is selected (between 0 and 5), and arguments are generated randomly to fill these positions. Note that in these cases there are no callback arguments generated: the default case is that the function takes no callback arguments.

For the non-callback arguments, their type is selected randomly from the JavaScript primitive types (number, string, and boolean), object literals, arrays, functions, and *other*. If the selected type is any of the primitives, `object`, or `array`, then these are fulfilled with randomly generated values of this type. Since we are working with many file system-related libraries, the random strings are selected from a pre-made list of valid file names, which correspond to a small hierarchy of directories and files created for the purposes of the testing. If the selected type is **`function`**, the

approach will select a random available function, either part of the API under test or part of the runtime environment (e.g., `console.log`).

If the selected type is *other*, this will reference a variable in the environment that has been generated from a previous call (and that remains in scope). This includes return values from previous API calls, and arguments to previous callbacks in the test case tree (see Definition 3).

5.3 Mining API Usages

Having an API call nested in the callback argument of another API call implies a connection between these calls. We define a notion of a *nesting example* to formalize the link between outer and inner functions and their arguments.

Definition 4 (Nesting example)**.** A *nesting example* is tuple

$$(f_{outer}, (arg_1^{outer}, ..., arg_m^{outer}), f_{inner}, (arg_1^{inner}, ..., arg_n^{inner}))$$

where

- f_{outer} is the name of a called API function,
- every arg_i^{outer} is either cb_{sync} or cb_{async} (meaning a sync or async callback argument) or _ (meaning any other non-callback argument),
- f_{inner} is the name of an API function invoked in the callback given to f_{outer}, and
- every arg_j^{inner} is either $outer@k$ (meaning the same argument as given to f_{outer} at position k), $cb@k$ (meaning the kth parameter of the callback function), or _ (meaning any other argument).

For example, the approach would mine the following nesting example from the nested function calls in Figure 1:

$$(ensureFile, (\_, \_, cb_{async}), readJson, (outer@0))$$

As another example, consider the following usage of the `fs-extra` API, where the `obj` parameter of the callback argument to `readJson` serves as an argument in a nested call to `outputJson`:

```
41  // read the contents of file.json and output it to output.json
42  readJson("file.json", function callback(err, obj) {
43     outputJson("output.json", obj);
44  });
```

The corresponding nesting example is:

$$(readJson, (\_, cb_{async}), outputJson, (cb@1, \_))$$

In order to determine what nesting examples exist in real-world use of the APIs, we design a static analysis to traverse the AST of existing API clients and identify such pairs. We implement this analysis in codeQL [GitHub 2021], making use of their extensive libraries for writing static analyzers. In particular, we use their access path tracking to identify functions as originating from an API import, and their single static assignment representation of local variables to identify shared arguments in the nestings. For shared arguments that are primitive values (e.g., the same string passed to both inner and outer API calls) we identify the link by checking for value equality.

The set of mined nesting examples is used as an input to the test generator. In *chooseFunction*, when selecting a function to be nested in the callback of some other function, the set of nesting examples is consulted to find examples where f_{outer} matches the outer function. If such nesting examples exist, then the algorithm randomly selects one of them and chooses the corresponding f_{inner} as the function to invoke inside the callback. Likewise, in *chooseArgument*, when generating arguments for the inner function, the selected nesting example is consulted to determine which

Table 1. Summary of projects used for evaluation.

Project	LOC	Stmts.	Cov. loading	Commit	Description
fs-extra	907	1.8k	16.8%	6bffcd8	Extra file system methods and promise support
jsonfile	45	102	19.1%	9c6478a	Read/write JSON files in Node.js
node-dir	285	496	5.9%	a57c3b1	Utility for common directory/file operations
bluebird	3.3k	5.6k	23.7%	6c8c069	Performance-oriented promise library
q	760	2.1k	22.2%	6bc7f52	Promise library
graceful-fs	439	825	25.3%	c1b3777	Drop-in replacement for native `fs`
rsvp.js	579	2.5k	16.4%	21e0c97	Tools for organizing async code
glob	845	1.5k	11.0%	8315c2d	Shell-style file pattern matching
zip-a-folder	24	48	16.0%	5089113	Zip/tar utility
memfs	2.4k	3.3k	29.1%	ec83e6f	In-memory file system, implements the Node.js `fs`

arguments (if any) to reuse from the outer function or the surrounding callback, and what position(s) they should take in the argument list.

Note that if there are no relevant mined nesting examples available, then the inner function is randomly selected. Also, the test generator is configured to only use the mined data 50% of the time, even for nestings where there exist some relevant mined nesting examples. If we only used nestings that showed up in the mined data, this would exclude many potentially correct pairs that simply do not occur in the mined projects. In Section 6, we explore the effect of varying how often the mined data should be consulted on the coverage achieved by the tests.

6 EVALUATION

We present our evaluation as a series of research questions.

- **RQ1**: How effective is the discovery phase at finding abstract signatures of API functions?
- **RQ2**: How effective is AsyncLambdaTester at achieving code coverage?
- **RQ3**: How effective is AsyncLambdaTester at finding behavioral differences during regression testing?
- **RQ4**: What is the effect (on coverage) of varying the chance of choosing nested function pairs based on the mined nesting examples?
- **RQ5**: What is the performance of AsyncLambdaTester?

Benchmarks. Table 1 shows the libraries we use as benchmarks for our evaluation, along with the number of lines of code (LOC), the number of statements[5], the number of statements covered by just importing the library (the *load bound*), the commit of the version of the library we used for evaluation, and a description of the library. We select popular JavaScript libraries that include some asynchronous behavior, such as libraries that access the file system and libraries to implement promises, i.e., objects that represent the result of an asynchronously executed operation. As a point of reference, we measure the statement coverage of the library code achieved by simply loading the library. The first row of this table can be read as follows: `fs-extra` source code has 907 LOC and 1.8k statements; the statement coverage from just importing the library is 16.8%; the last two columns are the tested version and the description. To mine nesting examples from existing API clients, we run the API usage mining over a corpus of 10,000 JavaScript projects on GitHub, which yields a set of 873 unique nestings of API functions in the libraries under test.[6]

Baselines and variants of the approach. We compare AsyncLambdaTester against the state-of-the-art approach LambdaTester [Selakovic et al. 2018], or short *LT*. Because the original LT does

[5] As computed by `nyc`, the coverage tool we use.

[6] The full list of nesting examples is included in the supplementary material.

Table 2. Abstract signatures categorized manually and found by the automated API discovery.

	Signatures found *with* callbacks				Signatures found *without* callbacks			
Project	**Manual**	**Autom.**	**Only manual**	**Only autom.**	**Manual**	**Autom.**	**Only manual**	**Only autom.**
`fs-extra`	21	88	3	70	45	361	21	337
`jsonfile`	4	8	0	4	8	12	4	8
`node-dir`	9	6	4	1	1	11	0	10
`bluebird`	25	22	7	4	29	68	26	65
`q`	16	27	7	18	57	155	19	117
`graceful-fs`	No docs	15	N/A	N/A	No docs	36	N/A	N/A
`rsvp.js`	3	7	0	4	10	31	3	24
`glob`	6	6	0	0	4	6	1	3
`zip-a-folder`	0	0	0	0	3	4	2	3
`memfs`	58	30	33	5	57	62	56	61

not support language features introduced in ECMAScript 6 and later, and because parts of the implementation are specific to their benchmarks, we re-implement LT within our testing framework. To better understand the value of nesting and sequencing calls to API functions, we evaluate two variants of AsyncLambdaTester: *ALT (seq)*, which uses sequencing only, and *ALT (seq+nest)*, which uses the full approach, i.e., including both sequencing and nesting.

6.1 RQ1: Effectiveness of Automated API Discovery

The following measures the effectiveness of AsyncLambdaTester at discovering the signatures of API functions. We inspect the documentation of the libraries and manually establish their signatures, and then compare these signatures against those discovered through our automated API discovery. As described in Section 4, if Algorithm 1 does not discover any valid signatures for an API function then a default "callback-less" signature is assigned. We do not count this default signature towards the total number of discovered signatures.

Table 2 displays the results of this experiment. For each API, we include the number of signatures found through the manual documentation inspection and the automated discovery, both with callback arguments and without callback arguments.[7] We also include the number only found with one of these approaches. The first row reads as follows: for `fs-extra`, manual analysis yields 21 signatures with callback arguments, and the automated discovery phase yields 88 signatures with callback arguments. Of those manually found, three are not found by the automated discovery; of those automatically discovered, 70 are not found by the manual analysis. The next four columns read the same way but for signatures without callback arguments. Note that we do not have results for `graceful-fs`, as it is undocumented.[8] Across all the documented APIs, our discovery phase finds 62% of the documented signatures with callback arguments, and 38% of those without.

Signatures found only manually. AsyncLambdaTester sometimes misses signatures because the automated discovery phase may fail to generate valid arguments, particularly in cases where the arguments need to meet specific conditions. For example, in the file system libraries, functions without callback arguments often expect valid file names for files with particular characteristics, and throw an error when this is not the case (such as `writeFileSync` and `readFileSync` in `jsonfile`). Another reason for signatures missed by the API discovery are functions that take *multiple* callback

[7]The full list of all manually and automatically discovered signatures for each API are included in the supplementary materials.

[8]This package is presented as a drop-in replacement for the native `fs` module, and so its documentation is basically "use it as you would the `fs` module".

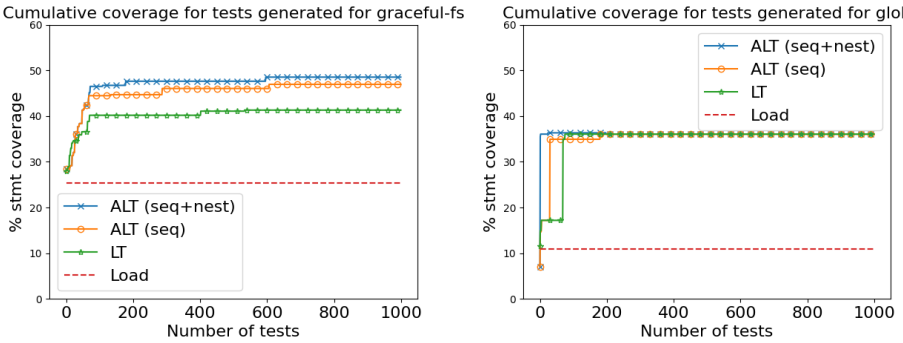

Fig. 4. Cumulative coverage while generating 1,000 tests for a library.

arguments, which our algorithm misses as it tests with only one callback at a time. Multiple callbacks are the main cause of missed signatures in `node-dir` and `memfs`.

Signatures found only automatically. There are two main reasons for finding signatures automatically that we missed during our manual inspection of documentation. First, some functions are undocumented, such as internal functions, aliases for the documented API functions, or re-exported functions of the built-in `fs` module. For example in `fs-extra`, `writeJson` can also be called with `writeJSON`. Since the automated discovery reads the function properties of the package on import, it tests all functions regardless of whether or not they are presented to the user in the documentation. There are also some internal functions that are present as properties on the library import, such as `_toUnixTimestamp` on `fs-extra` and `memfs`. Second, some API functions support more function signatures than those that are documented. Since the discovery phase only considers a signature invalid if the API function call throws an error with the tested arguments, a basic lack of error checking in the implementation can lead to many extra signatures. For example, consider the `writeFile` function in the `jsonfile` library. The documentation represents its signature as: `writeFile(filename, obj, [options], callback)`. However, the automated discovery phase finds that (*async*) is a valid signature. This unexpected signature is because `jsonfile.writeFile` is implemented with `universalify`'s `fromPromise` function, which executes the last argument if it is a callback, regardless of the other arguments. We made pull requests addressing this issue on both `fs-extra` and `jsonfile`.[9]

Answer to RQ1: The automated discovery finds 62% of the documented signatures that expect callback arguments, and 38% of API signatures without callback arguments. The approach also discovers some undocumented signatures, which in several cases are unexpected behavior.

6.2 RQ2: Coverage Achieved by Generated Tests

To measure how effective AsyncLambdaTester is at covering the statements of the library under test, we generate 1,000 tests for each library and compute the cumulative coverage. To compute coverage, we use the Istanbul command line coverage tool `nyc`, even if the developers include their own command for computing coverage, to ensure consistency. We repeat this experiment for the two variants of AsyncLambdaTester, ALT (seq+nest) and ALT(seq), and for the baseline LT approach.

[9]Links omitted for double-blind reviewing.

Table 3. Coverage after 1,000 tests and comparison with LambdaTester (LT).

	Coverage after 1,000 tests			Tests to match (exceed) 1,000 LT tests	
Project	**ALT (seq+nest)**	**ALT (seq)**	**LT**	**ALT (seq+nest)**	**ALT (seq)**
`fs-extra`	37.2%	35.4%	34.4%	311 (311)	663 (663)
`jsonfile`	87.2%	80.9%	80.9%	107 (209)	95 (N/A)
`node-dir`	32.5%	32.5%	32.5%	96 (N/A)	115 (N/A)
`bluebird`	48.0%	47.1%	47.1%	433 (433)	644 (N/A)
`q`	67.2%	66.4%	67.1%	103 (105)	765 (765)
`graceful-fs`	48.5%	47.0%	41.3%	49 (53)	49 (53)
`rsvp.js`	66.0%	66.0%	64.8%	177 (268)	196 (268)
`glob`	36.1%	36.1%	36.1%	3 (25)	181 (N/A)
`zip-a-folder`	32.0%	24.0%	24.0%	1 (793)	3 (N/A)
`memfs`	55.5%	48.3%	48.3%	84 (84)	702 (N/A)

Figure 4 shows how the cumulative coverage evolves while generating 1,000 tests. We show results for two representative libraries and provide the remaining plots in the supplementary material. As a reference, the horizontal line shows the load bound, i.e., coverage directly after loading the library. The coverage charts follow the same general logarithmic shape: a steep increase in coverage with the initial tests and an eventual convergence to some coverage plateau, or at least, a leveling off of the curve. The final coverage is fairly close between the two variants of AsyncLambdaTester, but the combination of sequencing and nesting converge more quickly. For `graceful-fs`, our approach achieves a final coverage much higher than LT, while with `glob` all three approaches converge to the same final coverage.

To quantify and summarize the coverage results for all libraries, the left part of Table 3 shows the final coverage achieved after 1,000 tests. The right part of the table quantifies the comparison with LT. We compute the number of tests required with AsyncLambdaTester to *match* and *exceed* the coverage that LT achieves after 1,000 tests. The first row can be read as follows: For the `fs-extra` project, after 1,000 tests AsyncLambdaTester achieves a statement coverage of 37.2% with ALT (seq+nest) and 35.4% with ALT (seq), while LT achieve 34.4%. ALT (seq+nest) matches and also exceeds LT's coverage after only 311 tests; ALT (seq) matches and also exceeds LT's coverage after 663 tests. Overall, AsyncLambdaTester consistently achieves slightly higher coverage than LT. Moreover, our approach reaches high coverage faster: It matches and often also exceeds the coverage that LT has after 1,000 tests with substantially fewer tests. Comparing the two variants of AsyncLambdaTester, the combination of sequencing and nesting is more effective.

Answer to RQ2: AsyncLambdaTester achieves a higher coverage than the state-of-the-art, and fewer tests are required to reach this coverage, in particular, when the approach uses both sequencing and nesting.

6.3 RQ3: Finding Behavioral Differences during Regression Testing

As a concrete application of AsyncLambdaTester, we apply the generated tests to check for behavioral differences in consecutive commits of the benchmark libraries.

6.3.1 Experimental Design. To compare the behavior of a library at two commits, we generate 100 tests based on the code at the earlier commit, and then run these tests with the code at both commits. We use the popular JavaScript testing framework `mocha`[10] to run our tests. This

[10] See https://mochajs.org/.

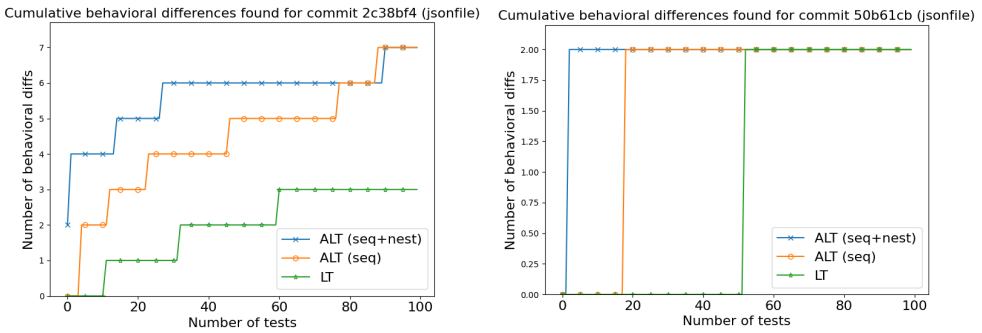

Fig. 5. Cumulative number of behavioral diffs, for specific commits for `jsonfile`.

framework can internally handle catastrophic errors in a test, so that all tests are executed even if an error is thrown in one of them. Most relevantly for us, `mocha` handles errors thrown by asynchronously invoked functions and logs this as an *internal async error*. To detect behavioral differences, AsyncLambdaTester produces the following output during test execution: values of all API function arguments before and after a call; the name of the API function a callback is passed to and the value of all parameters inside a callback being executed; the return value of a successful API function call; the name of the API function in the event of a failing call. We compare the outputs of both commits to identify the following *kinds of behavioral differences*:

- An API call resulting in an error in one commit successfully executes in the other.
- A return value of an API function call differs between commits.
- An argument to an API function call or a parameter of a callback differs between commits.
- A callback is called in one commit but not in the other.
- `mocha` reports an internal async error in one commit but not in the other.

Some API updates result in an output difference that does *not* indicate a relevant difference in functionality, e.g., due to function renaming, a change in the supported version of Node.js, or syntax errors resulting from migrating to `strict` mode. After manually identifying these cases, we configure our analysis to ignore them and do not count them in the experimental results.

To avoid double-counting the same behavioral difference being exposed by multiple generated tests, we consider differences as *equivalent* if they are due to the same kind of difference and arise from the same API function. Since we are working with asynchronous APIs, the exact ordering of calls may be non-deterministic, possibly causing different outputs across repeated executions of the same test. To avoid reporting scheduling differences as behavioral differences we execute each test ten times for the same commit and then compare the sets of observed outputs across commits. We report a behavioral difference only if an output observed in one commit is never observed in the other.

6.3.2 Quantitative Results. We quantitatively report the results of this experiment in three ways. At first, Figure 5 shows the cumulative number of detected behavioral differences while generating 100 tests for two particular commits of `jsonfile`. These commits were chosen arbitrarily from the set of commits for which AsyncLambdaTester finds behavioral differences. In both cases, the variant of our approach with both sequencing and nesting is most successful, followed by the sequencing-only variant, and then the existing LT technique.

To summarize these results across different commits, we compute the cumulative results per commit and then sum up the total number of behavioral differences across all commits of a library

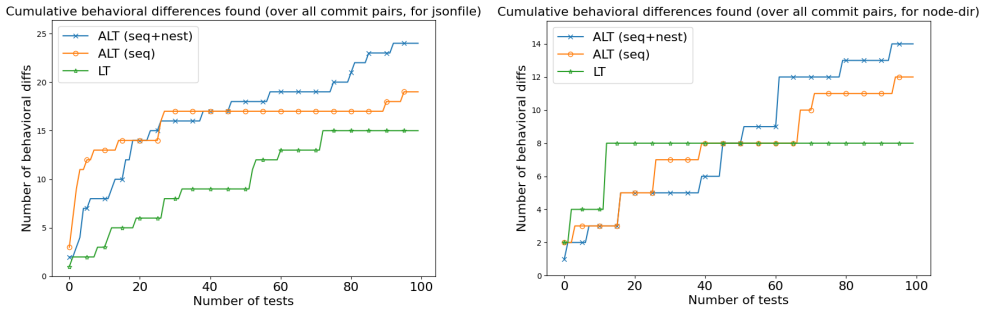

Fig. 6. Cumulative number of behavioral diffs summed over all commits.

Table 4. Pairs of commits checked via regression testing and behavioral differences found.

		With diff. (avg. unique per diff)		
Project	**Compared pairs**	**ALT (seq+nest)**	**ALT (seq)**	**LT**
`fs-extra`	99	17 (2.8)	14 (3.2)	10 (3.2)
`jsonfile`	44	12 (1.9)	9 (2.0)	8 (1.9)
`node-dir`	29	8 (1.5)	6 (2.3)	5 (1.6)
`bluebird`	99	0 (0.0)	0 (0.0)	0 (0.0)
`q`	99	92 (1.7)	69 (1.7)	61 (1.8)
`graceful-fs`	75	18 (1.1)	18 (1.1)	3 (1.3)
`rsvp.js`	99	0 (0.0)	0 (0.0)	0 (0.0)
`glob`	99	4 (1.5)	4 (1.3)	4 (1.0)
`zip-a-folder`	6	1 (1.0)	1 (1.0)	1 (1.0)
`memfs`	99	14 (1.1)	14 (1.1)	0 (0.0)

found after a specific number of generated tests. We display these graphs for `jsonfile` and `node-dir` in Figure 6. Again, the graphs for the other libraries are in the supplementary material.

Table 4 summarizes and quantifies these results across all libraries. For each library, we display the number of commit pairs being compared. We aim to compare 100 commits (i.e., 99 pairs) per library, but some of the repositories have less than 100 commits that affect source files. The table reports the number of commit pairs in which the approach detects a behavioral difference, with the average number of unique (i.e., non-equivalent) differences in parentheses. For example, the first row of this table reads: For `fs-extra`, we run the regression testing over 99 pairs of commits. ALT (seq+nest) spots a behavioral difference in 17 of these pairs, with 2.8 unique differences per pair, on average. ALT (seq) finds a difference in 14 of these pairs (3.2 unique differences, on average), and LT in 10 of these (3.2 unique differences, on average). The results show the same trend as that seen in RQ2: AsyncLambdaTester finds more behavioral differences than LT, and the combination of nesting and sequencing is worthwhile.

6.3.3 Qualitative Results. To better understand the behavioral differences that AsyncLambdaTester reveals, we manually inspect some of them. Table 5 summarizes the results. For each of the differences analyzed, we include a hyperlink to the commit introducing the difference, the name of the project, a categorization of the difference, and a description of how it manifests. The categorizations we use are as follows:

- **Bug**: This commit is introducing a bug.
- **Bug fix**: This commit is fixing a bug.

Table 5. Manual analysis of behavioral differences found via regression testing.

Commit	Project	Diagnosis	Description of behavioral difference
a149f82	fs-extra	Bug	outputJSON executes callback even with bad arguments in newer commit
dba0cbb	fs-extra	Upgrade	many API functions no longer error or return different values in newer commit
03b2080	fs-extra	Bug fix	exists returns callback argument return value on error instead of undefined
df125be	fs-extra	Bug	ensureSymLink executes the callback argument even with incorrect arguments
3fc5894	fs-extra	Bug fix	ensureFile throws error on incorrect arguments instead of executing callback
ef9ade4	fs-extra	Bug	copyFile doesn't throw error with incorrect non-callback arguments
2e4fcae	fs-extra	Upgrade	writev returns rejected promise instead of throwing error on incorrect arguments
075c2d1	fs-extra	Bug	move and copy now executes the callback argument even with incorrect arguments
eaab0ee	glob	Upgrade	glob succeeds in newer commit (error in older commit)
2a9a617	graceful-fs	Bug fix	createReadStream succeeds in newer commit (infinite loops in older commit)
45a0242	graceful-fs	Bug fix	lchmod is undefined on Linux in older commit; succeeds in newer commit
5d961ab	graceful-fs	Bug fix	readFile sometimes executes callback in newer commit
4a0ebe5	jsonfile	Bug fix	readFile executes callback in newer commit
b1f40ef	jsonfile	Upgrade	readFileSync succeeds in newer commit (errors in older commit)
4b90419	jsonfile	Upgrade	readFileSync errors in newer commit (succeeds in older commit)
995aa63	jsonfile	Bug	writeFile executes callback in newer commit
e3d86e0	jsonfile	Bug	read/writeFile execute callback even with bad arguments in newer commit
10eed1d	jsonfile	Bug	readFile sometimes errors in newer commit (succeeds in older commit)
e5e5aa9	q	Upgrade	tap and any succeeds in newer commit (error in older commit)
5d8a060	zip-a-folder	Upgrade	zipFolder executes callback in newer commit

- **API upgrade**: This commit is an update/upgrade of the API, including migration to newer APIs, updating method signatures, and making functions **async**.

We find that AsyncLambdaTester detects a variety of different types of API functionality changes. Interestingly, for several commits that introduce a bug, AsyncLambdaTester later finds the "dual" commit that fixes that same bug.

Answer to RQ3: AsyncLambdaTester finds many behavioral differences between versions of libraries, including accidentally introduced bugs, bug fixes, and API upgrades. These differences are found most quickly with generated tests that use both sequencing and nesting.

6.4 RQ4: Impact of Guidance by Mined Nesting Examples

The API usage mining component of AsyncLambdaTester informs the choice of which inner API function to call when nesting API functions. By default, the mined nesting examples are consulted 50% of the time. The following measures the effect of varying this percentage on the coverage achieved by the generated tests. The experimental setup is as in RQ2, but we consider the test generation to follow mined nesting examples 0%, 25%, 50%, 75%, and 100% of the time.

Table 6 summarizes the result of this experiment by showing the final coverage after generating 1,000 tests. The corresponding coverage plots are in the supplementary material. The first row of this table reads as follows: For fs-extra, the statement coverage when using 0% mined nestings is 32.16%, when using 25% mined nestings it is 33.55%, when using 50% mined nestings it is 37.18%, when using 75% mined nestings it is 33.01%, and when using 100% mined nestings it is 33.01%. For readability, we show the highest coverage for each project in bold.

The results illustrate the value of using mined nesting examples: 50% mined nestings always leads to higher coverage than 0% and 100%. This supports our initial hypothesis that choosing informed nestings is more likely to produce valid tests, which will therefore increase the coverage

Table 6. Coverage after 1,000 tests at different levels of using mined nesting examples.

	Test coverage, using % mined nestings				
Project	**0%**	**25%**	**50%**	**75%**	**100%**
fs-extra	32.2%	33.6%	**37.2%**	33.0%	33.0%
jsonfile	**87.2%**	83.0%	**87.2%**	**87.2%**	**87.2%**
node-dir	32.5%	**33.6%**	32.5%	33.2%	33.2%
bluebird	45.7%	51.2%	48.0%	**55.1%**	48.4%
q	65.7%	66.4%	**67.2%**	66.3%	66.3%
graceful-fs	**48.5%**	**48.5%**	**48.5%**	**48.5%**	47.0%
rsvp.js	64.8%	65.0%	**66.0%**	58.2%	58.2%
glob	**36.1%**	**36.1%**	**36.1%**	**36.1%**	**36.1%**
zip-a-folder	24.0%	24.0%	**32.0%**	**32.0%**	24.0%
memfs	50.0 %	51.2%	**55.5%**	51.8%	51.8%

Table 7. Time to generate 100 tests.

	Time in seconds to generate 100 tests		
Project	**ALT (seq+nest)**	**ALT (seq)**	**LT**
fs-extra	19.7	19.0	19.3
jsonfile	17.2	17.4	17.3
node-dir	18.2	16.2	16.1
bluebird	25.5	24.4	24.5
q	16.7	17.4	17.4
graceful-fs	16.8	17.4	17.2
rsvp.js	24.0	24.4	24.1
glob	15.8	15.8	15.9
zip-a-folder	28.6	24.9	24.8
memfs	31.3	32.3	32.1

achieved. However, choosing only mined nestings, i.e, 100%, risks to miss valid nestings that are simply never seen during the API usage mining.

Answer to RQ4: Choosing mined nestings *some* of the time results in tests with higher coverage than those generated *always* or *never* using the mined nestings. The optimal parameter depends on the library under test, but 50% is an overall reasonable choice.

6.5 RQ5: Performance of Test Generation

Table 7 displays the time taken to generate 100 tests for each of three approaches we consider. We run these experiments on a machine with two 32-core 2.35GHz AMD EPYC 7452 CPUs and 128GB RAM, running CentOS 7.8.2003 and Node.js 14.16.1. Since AsyncLambdaTester is implemented in TypeScript, it is single-threaded and so only uses one core. The results show that the time required to generate 100 tests is fairly similar across all three approaches. Depending on the library, the test generation takes between 15 and 30 seconds.

Answer to RQ5: The two variants of AsyncLambdaTester, and also the baseline technique LT, take about the same amount of time to generate tests. With 15 to 30 seconds per 100 tests, the approach is efficient enough for practical use.

7 RELATED WORK

7.1 Test Generation for Higher-Order Functions

There are several techniques for automatically testing higher-order functions. Most closely related is LambdaTester [Selakovic et al. 2018], which also targets JavaScript and has inspired some of our design decisions. The main difference is that AsyncLambdaTester tests functions with both asynchronous and synchronous callbacks, enabled by our method for API discovery and through the notion of a test case tree, which allows for nesting calls. Our evaluation empirically compares with a re-implementation of LambdaTester and shows that AsyncLambdaTester reaches the same or even better coverage substantially faster, and reveals behavioral differences missed by LambdaTester.

Other test generators can be roughly categorized into random testing and solver-based, systematic testing. QuickCheck [Claessen and Hughes 2000] is an example of the former, as it creates callback functions that return a random, type-correct value. Koopman and Plasmeijer [2006] propose systematic, syntax-driven generation of callback functions based on user-provided generators. A test generator for higher-order functions in Racket relies on types and contracts of tested functions [Klein et al. 2010], two kinds of information that rarely exist for JavaScript libraries. Solver-based test generators include several variants of symbolic and concolic execution adapted to higher-order functions [Nguyen and Horn 2015; You et al. 2021], and work that performs a type-directed, enumerative search over the space of test cases [Song et al. 2019]. Palka et al. [2011] propose to randomly generate type-correct Haskell programs, including higher-order functions, to test a Haskell compiler. All of the above approaches target functional languages, and none of them considers asynchronous callbacks or produces nested callbacks.

7.2 Random Test Generation

AsyncLambdaTester builds upon a rich history of random test generators, starting with Randoop [Pacheco et al. 2007], which introduced feedback-directed random test generation. Our work also follows this paradigm, but in contrast to Randoop, addresses the challenges of higher-order functions and those arising in a dynamically typed language. EvoSuite [Fraser and Arcuri 2011] uses an evolutionary algorithm to continuously improve randomly generated test cases. Beyond function-level testing, application-level fuzzing has received significant attention, including AFL[11] and its derivatives [Böhme et al. 2019; Lemieux and Sen 2018], and combinations of fuzzing with symbolic testing [Stephens et al. 2016]. In contrast to the above greybox or whitebox fuzzers, AsyncLambdaTester does not need to analyze the library under test, but obtains feedback from the execution of the generated tests alone.

7.3 Asynchronous JavaScript

A study of callbacks in JavaScript code finds that 10% of all functions take callback arguments, that the majority of those callbacks are nested, and that the majority of callbacks are asynchronous [Gallaba et al. 2015]. These results show that generating tests without considering asynchronous callbacks (i.e., the tests that prior work [Selakovic et al. 2018] is able to generate), fails to fully reflect the behavior seen in real-world JavaScript code. Another study reports that most of the concurrency bugs in Node.js are about usages of asynchronous APIs [Wang et al. 2017]. Our work is about analyzing the implementation of such APIs. Beyond JavaScript, a study of higher-order functions in Scala finds that 7% of all functions are higher-order [Xu et al. 2020], suggesting that the problem we address is relevant beyond JavaScript.

Alimadadi et al. [2016] propose a dynamic analysis to trace and visualize JavaScript executions, with a focus on asynchronous interactions across the client and the server. Another dynamic

[11] https://lcamtuf.coredump.cx/afl/

analysis detects promise-related anti-patterns [Alimadadi et al. 2018]. Several techniques aim at detecting races in JavaScript [Adamsen et al. 2018, 2017; Petrov et al. 2012; Raychev et al. 2013; Zheng et al. 2011], where "race" means that different asynchronously scheduled callbacks may be executed in more than one order. These approaches are also motivated by the challenges of asynchronous JavaScript but address problems orthogonal to that addressed by AsyncLambdaTester.

There are several formalizations of different aspects of asynchronous JavaScript, including an execution model of Node.js [Loring et al. 2017], a model to reason about promises [Madsen et al. 2017], and a calculus, semantics, and implementation of a static analysis of asynchronous behavior [Sotiropoulos and Livshits 2019]. The "callback graph" of the latter relates to our test case trees, but it is created as part of a static analysis and captures a happens-before-like relation, while test case trees serve as an intermediate representation that represents sequencing and nesting.

7.4 Program Analysis for JavaScript

The popularity of JavaScript has motivated a variety of dynamic and static analysis techniques, and we refer to a survey for a comprehensive discussion [Andreasen et al. 2017]. Examples of techniques include dynamic analyses to detect type inconsistencies [Pradel et al. 2015], to detect inefficient code [Gong et al. 2015a], to detect various common programming mistakes [Gong et al. 2015b], and to reason about taint flows [Karim et al. 2020]. Work on reasoning about API changes and how they affect clients [Møller et al. 2020] is a recent example of a static analysis. Similar to AsyncLambdaTester, all these analyses take a pragmatic approach toward addressing the idiosyncrasies of JavaScript, without providing strong soundness or completeness guarantees.

8 CONCLUSION

Effective test generation for APIs that make use of asynchronous callback arguments is challenging, as the test generator must generate tests that combine multiple calls to related API functions. Generating only sequences of calls, as done in existing test generators, is insufficient. In this paper we presented AsyncLambdaTester, the first test generator aimed at APIs with asynchronous callbacks, which addresses the above challenges by both sequencing and nesting API calls. Nesting here means that when a callback is invoked asynchronously, it triggers more API calls, which can use values available only once the callback has been invoked. We applied AsyncLambdaTester to ten popular JavaScript libraries that contain 142 API functions with callbacks, and measured the effectiveness of the generated tests over a variety of metrics. We find that AsyncLambdaTester achieves high coverage (between 32% and 87%) and detects various behavioral differences in a regression testing scenario, some of which correspond to bugs the developers introduced without noticing it. Compared to a state-of-the-art tool, AsyncLambdaTester exercises otherwise missed behavior and reaches interesting behavior faster.

REFERENCES

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*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 38–48. https://doi.org/10.1145/3236024.3236038

Christoffer Quist Adamsen, Anders Møller, and Frank Tip. 2017. Practical initialization race detection for JavaScript web applications. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 66:1–66:22. https://doi.org/10.1145/3133890

Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2016. Understanding asynchronous interactions in full-stack JavaScript. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 1169–1180.

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

Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A Survey of Dynamic Analysis and Test Generation for JavaScript. *Comput. Surveys* (2017).

Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Trans. Software Eng.* 45, 5 (2019), 489–506. https://doi.org/10.1109/TSE.2017.2785841

Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. 2008. ARTOO: adaptive random testing for object-oriented software. In *International Conference on Software Engineering (ICSE)*. ACM, 71–80.

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, 268–279. https://doi.org/10.1145/351240.351266

Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: an automatic robustness tester for Java. *Software Prac. Experience* 34, 11 (2004), 1025–1050.

Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT-/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. 416–419.

Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. 2015. Don't Call Us, We'll Call You: Characterizing Callbacks in Javascript. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2015, Beijing, China, October 22-23, 2015*. 247–256.

GitHub. 2021. QL standard libraries. https://github.com/Semmle/ql. Accessed: 2021-04-13.

Liang Gong, Michael Pradel, and Koushik Sen. 2015a. JITProf: Pinpointing JIT-unfriendly JavaScript Code. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 357–368.

Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015b. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *International Symposium on Software Testing and Analysis (ISSTA)*. 94–105.

Rezwana Karim, Frank Tip, Alena Sochurková, and Koushik Sen. 2020. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Trans. Software Eng.* 46, 12 (2020), 1364–1379. https://doi.org/10.1109/TSE.2018.2878020

Casey Klein, Matthew Flatt, and Robert Bruce Findler. 2010. Random testing for higher-order, stateful programs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 555–566. https://doi.org/10.1145/1869459.1869505

Pieter W. M. Koopman and Rinus Plasmeijer. 2006. Automatic Testing of Higher Order Functions. In *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4279)*, Naoki Kobayashi (Ed.). Springer, 148–164. https://doi.org/10.1007/11924661_9

Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 475–485.

Matthew C. Loring, Mark Marron, and Daan Leijen. 2017. Semantics of asynchronous JavaScript. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages, Vancouver, BC, Canada, October 23 - 27, 2017*, Davide Ancona (Ed.). ACM, 51–62. https://doi.org/10.1145/3133841.3133846

Magnus Madsen, Ondrej Lhoták, and Frank Tip. 2017. A model for reasoning about JavaScript promises. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 86:1–86:24. https://doi.org/10.1145/3133910

Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. 2020. Detecting locations in JavaScript programs affected by breaking library changes. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 187:1–187:25. https://doi.org/10.1145/3428255

Phuc C. Nguyen and David Van Horn. 2015. Relatively complete counterexamples for higher-order programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 446–456.

Semih Okur, David L Hartveld, Danny Dig, and Arie van Deursen. 2014. A study and toolkit for asynchronous programming in C#. In *Proceedings of the 36th International Conference on Software Engineering*. 1117–1127.

Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. 2008. Finding errors in .NET with feedback-directed random testing. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 87–96.

Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *International Conference on Software Engineering (ICSE)*. IEEE, 75–84.

Michal H. Palka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test, AST 2011, Waikiki, Honolulu, HI, USA, May 23-24, 2011*, Antonia Bertolino, Howard Foster, and J. Jenny Li (Eds.). ACM, 91–97. https://doi.org/10.1145/1982595.1982615

Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. 2012. Race Detection for Web Applications. In *Conference on Programming Language Design and Implementation (PLDI)*.

Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript. In *International Conference on Software Engineering (ICSE)*.

Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective Race Detection for Event-Driven Programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Marija Selakovic, Michael Pradel, Rezwana Karim, and Frank Tip. 2018. Test generation for higher-order functions in dynamic languages. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 263–272.

Dowon Song, Myungho Lee, and Hakjoo Oh. 2019. Automatic and scalable detection of logical errors in functional programming assignments. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 188:1–188:30. https://doi.org/10.1145/3360614

Thodoris Sotiropoulos and Benjamin Livshits. 2019. Static Analysis for Asynchronous JavaScript Programs. *CoRR* abs/1901.03575 (2019). arXiv:1901.03575 http://arxiv.org/abs/1901.03575

Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*.

Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. 2004. Test input generation with Java PathFinder. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 97–107.

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.

Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. 2005. Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 365–381.

Yisen Xu, Fan Wu, Xiangyang Jia, Lingbo Li, and Jifeng Xuan. 2020. Mining the use of higher-order functions. *Empir. Softw. Eng.* 25, 6 (2020), 4547–4584. https://doi.org/10.1007/s10664-020-09842-7

Shu-Hung You, Robert Bruce Findler, and Christos Dimoulas. 2021. Sound and Complete Concolic Testing for Higher-order Functions. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 635–663. https://doi.org/10.1007/978-3-030-72019-3_23

Yunhui Zheng, Tao Bao, and Xiangyu Zhang. 2011. Statically locating web application bugs caused by asynchronous calls.. In *WWW*. 805–814.