Nessie: Automatically Testing JavaScript APIs with Asynchronous Callbacks

Ellen Arteca
arteca.e@northeastern.edu
Northeastern University
USA

Sebastian Harner
harnersebastian@gmail.com
University of Stuttgart
Germany

Michael Pradel
michael@binaervarianz.de
University of Stuttgart
Germany

Frank Tip
f.tip@northeastern.edu
Northeastern University
USA

ABSTRACT

Previous algorithms for feedback-directed unit test generation iteratively create sequences of API calls by executing partial tests and by adding new API calls at the end of the test. These algorithms are challenged by a popular class of APIs: higher-order functions that receive callback arguments, which often are invoked asynchronously. Existing test generators cannot effectively test such APIs because they only *sequence* API calls, but do not *nest* one call into the callback function of another. This paper presents Nessie, the first feedback-directed unit test generator that supports nesting of API calls and that tests asynchronous callbacks. Nesting API calls enables a test to use values produced by an API that are available only once a callback has been invoked, and is often necessary to ensure that methods are invoked in a specific order. The core contributions of our approach are a tree-based representation of unit tests with callbacks and a novel algorithm to iteratively generate such tests in a feedback-directed manner. We evaluate our approach on ten popular JavaScript libraries with both asynchronous and synchronous callbacks. The results show that, in a comparison with LambdaTester, a state of the art test generation technique that only considers sequencing of method calls, Nessie finds more behavioral differences and achieves slightly higher coverage. Notably, Nessie needs to generate significantly fewer tests to achieve and exceed the coverage achieved by the state of the art.

ACM Reference Format:
Ellen Arteca, Sebastian Harner, Michael Pradel, and Frank Tip. 2022. Nessie: Automatically Testing JavaScript APIs with Asynchronous Callbacks. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510106

1 INTRODUCTION

Test generation is an important technique to automatically test libraries by creating unit-level tests. The generated tests typically consist of a sequence of calls to functions in an API under test. The values passed as arguments to each function call in such a sequence may be chosen randomly, or values returned by previous calls in the sequence can be used, to facilitate 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 [10, 12], feedback from executions [13, 29, 30], and symbolic reasoning [36, 40, 42].

However, existing work on test generation has ignored a broad class of APIs: Functions that accept another function as a callback argument and then invoke that other function asynchronously. The key benefit of asynchronous callbacks is that they do 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 [14], but also prone to mistakes and surprising behavior [28, 41]. While the JavaScript community has started migrating to asynchronous APIs that rely on promises [1, Section 25.6] and `async`/`await` [1, Section 25.7], a vast amount of JavaScript code still uses event-driven APIs that invoke callback functions passed to them asynchronously.

We observe that existing test generators miss out on two opportunities for testing APIs with asynchronous callbacks. First, in addition to *sequencing* function calls, one can also consider *nesting* them, by placing an API call into a callback passed to another API function under test. Such nesting enables a test to use values produced by an API that are available only once a callback has been invoked; moreover, nesting is often necessary to ensure a specific ordering of invocations to API functions. Second, even the best existing test generator aimed at testing functions with callbacks [35] supports only *synchronous*, but not *asynchronous* callbacks. The table below compares our work with the capabilities of two related techniques our work is inspired by:

	Sequencing	Nesting	Synchronous callbacks	Asynchronous callbacks
Randoop [30]	✓			
LambdaTester [35]	✓		✓	
This work (Nessie)	✓	✓	✓	✓

To illustrate the challenges associated with testing asynchronous APIs, consider a library for accessing JSON files. The API defines a function `exists` for checking whether a JSON file with a specified name exists, which produces a handle to that file if this is the case. Moreover, there is a function `read` for parsing a JSON file

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510106

represented by a given handle. In JavaScript, functions in event-driven APIs typically take a callback as their last argument, which is invoked with two arguments: (i) an object that indicates whether an error has occurred (or `null` if no error occurred) and (ii) an object representing the results of the operation. Hence, a typical use of the library would look as follows:

```
let fileName = ...;
exists(fileName,
  // asynchronously invoked callback function:
  (err, fileHandle) => {
    if (!err) {
      read(fileHandle,
        // another callback function:
        (err, jsonObj) => { ... })
    }
})
```

The call to `read` is nested in the callback that is passed to `exists`, to ensure that the read operation is executed after the exists operation has completed. Now suppose that the `read` operation contains a bug that is triggered in certain cases where a valid file-handle is passed (e.g., if the file's permissions do not permit read access), and suppose that we want to generate a test that invokes the `read` function to expose the bug. Since file-handle objects are created inside the library, it is unclear what the representation of these objects looks like without analyzing or executing the library code. While it is possible for a test generator to create suitable file-handle objects using a purely random approach, the chances of successfully creating a valid file-handle would be small. Therefore, the most effective way to obtain a valid file-handle and expose the bug is to invoke `exists` with some callback function f, and invoke `read` with the file-handle that is passed to f as its second argument. That is, we would generate a test where a call to `read` is *nested* in the callback that is passed to `exists`, as in the above example.

Unfortunately, the state of the art test generator for testing functions with callbacks [35] is unable to generate such a test for the two reasons mentioned in the above table: First, it fails to identify API functions that receive an asynchronously invoked callback argument, and hence, never passes callbacks into such functions. Second, it does not construct tests where a call to one API function is nested inside the callback passed to another API function.

This paper presents Nessie, the first feedback-directed test generation technique that nests API calls into callbacks and that supports 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. We present an algorithm for iteratively generating tree-shaped tests based on feedback from executing already generated tests. The algorithm is supported by an automated API discovery phase that determines which API functions accept asynchronous or synchronous callbacks and by guiding the test generator toward realistic API usages based on nesting examples mined from existing API clients.

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. Our evaluation applies Nessie to ten popular JavaScript libraries that include a total of 356 API functions, 142 of which expect callbacks. Nessie's API discovery phase detects 62% of the API signatures with callback arguments that are mentioned in the API's documentation, and 106 *undocumented* API signatures with callbacks, reflecting unexpected behavior. The coverage achieved by Nessie converges significantly more quickly than with the state of the art LambdaTester approach [35] and even reaches a slightly higher coverage: On average, Nessie needs to generate only 136 tests to achieve the same coverage that LambdaTester achieves with 1,000 generated tests. We conjecture that this is the case because the nesting of callback functions enables a more effective selection of argument values in subsequent (nested) function calls. We also compare the ability of Nessie and LambdaTester to detect situations where tests generated for a given version of an API behave differently when run against the next version of that API. On average, Nessie detects 23% more behavioral differences than LambdaTester, including a mix of bugs and intentional API changes. While these differences can, in principle, all be detected without nesting API calls, our approach finds them more effectively and efficiently due to its ability to nest calls.

In summary, this paper contributes the following:

- The first automated test generator specifically aimed at APIs that accept callbacks to be invoked asynchronously.
- An algorithm for incrementally generating tests that not only sequence API calls but also nest them inside callbacks.
- Empirical evidence demonstrating that: (i) the approach is effective at exercising JavaScript APIs with asynchronous callbacks, (ii) that it achieves modestly higher code coverage and finds more behavioral differences than the state of the art, and (iii) that it *converges much more quickly* than prior work when it comes to achieving a specific level of coverage or behavioral differences.

2 OVERVIEW OF NESSIE

This paper presents a test generation technique for testing higher-order functions. A function is called a *higher-order function* if it expects another function to be passed as an argument, or if it returns a function. Our work targets functions f that receive a callback function cb as an argument and then invoke the callback either synchronously or asynchronously. Asynchronous invocation here means that the execution of f causes cb to be invoked from the main event loop at some later time.

Generating tests for asynchronous APIs involves several open challenges. The first challenge is to find out which API functions expect (a)synchronous callbacks and at what argument positions these callbacks should be passed. Since JavaScript is dynamically typed, our approach needs to infer the signatures of functions as a prerequisite to generating effective tests. 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 third challenge is about how to compose API calls in a realistic way. To increase the chances that our approach nests API calls in ways that represent real-world API usages, it requires some knowledge of typical API usages.

Our approach, illustrated in Figure 1, consists of three main components that each address one of the challenges described above. Given a set of API functions to test, the first step is an *automated*

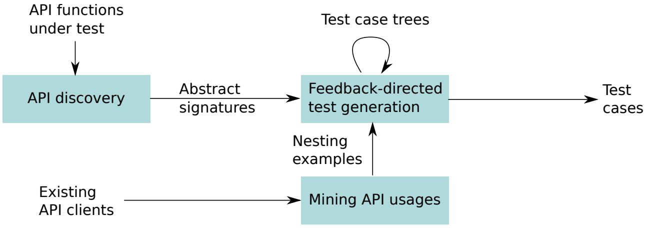


Figure 1: Overview of the approach.

API discovery, which probes the API functions under test to determine if and where they expect callback arguments. The discovered information is stored as a set of *abstract signatures*, which record whether the arguments are: (i) a synchronously invoked callback, (ii) an asynchronously invoked callback, or (iii) a value that is not a callback[1]. 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.

3 API DISCOVERY

The first step in test generation is determining what to test: Given a library under test, Nessie retrieves the set of all function properties offered by the library object. As JavaScript is a dynamically typed language, Nessie initially does not know anything about these functions beyond their names. One possible approach to learn more about the APIs would be to rely on optional type annotations, e.g., in the form of third-party TypeScript type declarations for popular libraries.[2] However, not all JavaScript code comes with type annotations [25] and even if it exists, an API's implementation may diverge from its declared type [21]. Nessie addresses this problem through an *API discovery* phase that probes API functions to determine at what parameter positions they expect callbacks and whether these are invoked synchronously or asynchronously. This information is recorded as a set of *abstract signatures*, or simply *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*: an argument is an asynchronously invoked callback
- *sync*: an argument is a synchronously invoked callback
- the _ symbol: any non-callback argument

A single function may have zero, one, or multiple signatures. Zero signatures indicates that the API discovery 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 when 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.[3] In our experience, such overloading is extremely common in JavaScript APIs due to optional arguments preceding the (final) callback argument.

To discover signatures for a given API function under test, Nessie repeatedly[4] invokes the function with randomly generated arguments. The approach alternates between generating calls with and without a callback argument, and passes different numbers of arguments. A generated test for a function `api` is structured as follows:

```
let callback = () => {console.log("Callback executed");}
try {
  // try calling the specified API function
  api(..., callback, ...);
  console.log("API call executed");
} catch(e) {
  console.log("Error in API call");
}
console.log("Test executed");
```

During the execution of these tests, the print statements track if and when callbacks are invoked, and whether the function terminates successfully. We ignore erroneous executions, as they may be due to incorrect arguments passed to the function. For non-erroneous executions, the approach distinguishes three cases:

- a callback that executes *after the test has executed* is executed asynchronously, so a signature is created with *async* at the position of the callback argument and _ at all other positions,
- a callback that executes *before the API call returns* is executed synchronously, so a signature is created with *sync* at the callback position and _ at all other positions,
- if the callback is not executed or the test does not pass any callback, a signature with _ symbols only is created.

We test with 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 to support multiple callbacks is straightforward but increases the computational complexity of API discovery. In general, the results of the API discovery phase are unsound, because the approach does not guarantee to cover all possible arguments or all paths through the API implementation.

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 any) used during test generation are informed by these discovered signatures.

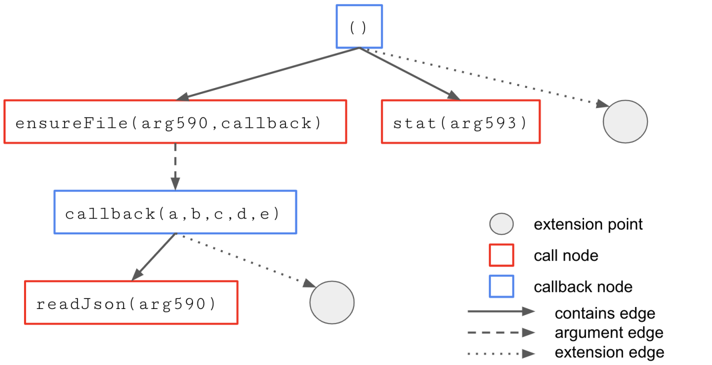
4 FEEDBACK-DIRECTED TEST GENERATION

The following describes the core algorithm of Nessie, which generates tests cases that both sequence and nest API calls. The algorithm is based on a tree-shaped representation of test cases called a *test case tree* (Section 4.1), which serves as the basis for the algorithm itself (Section 4.2).

[1] Since we are targeting JavaScript, where functions can be invoked with any number of arguments of any type, we do not attempt to track precise type information.
[2] https://definitelytyped.org/
[3] https://github.com/jprichardson/node-fs-extra/blob/HEAD/docs/outputJson.md
[4] By default Nessie runs 50 tests per API function, each with a timeout of 2 seconds.

```
1   // Omitted for clarity:
2   //  * Try-catch around each call to an API function.
3   //  * Print statements to log arguments and return values.
4   //  * Print statements to log control flow.
5
6   let fs_extra = require("fs-extra");
7
8   var arg590 = "a/b/test";
9   var arg593 = null;
10
11  let r_126_0 = fs_extra.ensureFile(arg590, cb(a, b, c, d, e) =>
12    let r_126_0_0 = fs_extra.readJson(arg590);
13    return false;
14  });
15
16  let r_126_1 = fs_extra.stat(arg593);
```

(a) Generated test.



(b) Test case tree representation of the test, with extension points annotated.

Figure 2: Example of a generated test for the API functions of the `fs-extra` library and the corresponding test case tree.

4.1 Test Case Trees

To represent test cases that support both sequencing and nesting of method calls, we define the *text case tree*, 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 a map of function names to abstract signatures. Then, a *test case tree* is an ordered tree where nodes are either:

- a *call node* of the form $r = api(a_1, ..., a_k)$, meaning that function api is invoked with arguments $(a_1, ..., a_k)$ and yields return value r. If $s_i \in \{sync, async\}$ for some signature $(api \mapsto (s_1, ..., s_n)) \in S$, then a_i may be a callback function. Otherwise, $a_i \in V$ or it is a return value of another call, or
- 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:

- a *contains edge* 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* 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 order of the call nodes under a callback node represents the sequential order in which calls occur in the body of a callback function.

Example. Figure 2 gives an example of a test case and its corresponding test case tree. Each API call in the test case in Figure 2a is represented by a call node in the test case tree of Figure 2b. The nodes for calls that receive callback arguments each have a corresponding callback node as a child. E.g., the callback given to `ensureFile` at line 11 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. E.g., the call to `readJson` on line 12 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 8 to 9.

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.

Example. Figure 2b shows a test case tree where 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 8), 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.

4.2 Test Generation Algorithm

Our algorithm creates test cases iteratively, by repeatedly extending an existing test case with another call at *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. For callback nodes with no children, there is an extension point for adding a first child node.

Algorithm 1 Feedback-directed generation.

```
Input: Set $F$ of API functions, map $S$ from function names in $F$ to
    their discovered signatures, mined nesting examples $M$
Output: Set of tests $T$
$T \leftarrow \emptyset$                                ▷ Generated test case trees
$t \leftarrow$ empty test case tree
$E \leftarrow \{(t, root(t))\}$                         ▷ Extension points
while $|T| <$ number of tests to generate do
    $(t, n) \leftarrow$randomly pick from $E$
    $f \leftarrow chooseFunction(t, n, F, M)$
    $args \leftarrow chooseArguments(t, n, f, S, M)$
    $t' \leftarrow$ extend $t$ with call to $f(args)$ at $n$
    $feedback \leftarrow execute(t')$
    $T \leftarrow T \cup \{t'\}$
    if $noExceptions(feedback)$ then
        for each $n' \in extensionPoints(feedback)$ do
            $E \leftarrow E\{(t', n')\}$
return T
```

Example. Figure 2b shows the extension points of the test case tree, as well as edges that would be added if the test is extended at these points (labeled the *extension edges*). These extension points correspond to adding a call right after lines 12 and 16 in Figure 2a.

Algorithm 1 summarizes our feedback-directed approach for creating test cases. It maintains a set T of generated tests and a set E of extension points. Each extension point is a pair (t, n) of a test case tree t and a node n in this tree, representing the callback node where a new call node could be inserted. The main loop extends an existing test with a new call at one of the extension points and adds the test to T. The extended test t' is then executed to gather feedback about its execution. If no exception is thrown, each possible callback node that is executed is kept as a possible extension point to create a further test. The algorithm relies on helper functions for choosing a function to call, choosing arguments to pass into the function, and identifying future extension points, which we describe below.

Choosing a function to call. Given a specific extension point, Algorithm 1 calls *chooseFunction* to pick a function to call by balancing two requirements. On the one hand, the generated tests should cover as many functions as possible. On the other hand, we do not want to prescribe a specific order in which 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*, its 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 4.3.

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 signatures produced by the discovery phase. If multiple signatures exist for a function, one is chosen at random. 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 no signatures exist for the function, 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, no callback arguments are generated.

For non-callback arguments, the 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 randomly generated values of this type are used. If the selected type is **`function`**, an available function is chosen from the API under test or the runtime environment (e.g., `console.log`). If the selected type is *other*, a variable that is available at the current scope is selected, which includes return values from previous API calls and arguments to previous callbacks in the test case tree (Definition 3).

Nesting API calls helps with generating well-formed arguments, as values (including objects) may get passed from outer to inner calls, as illustrated in the motivating example in Section 1. In addition, since we are working with many file system-related libraries, string primitive values are selected randomly from a pre-made list of valid file names that correspond to a small hierarchy of directories and files generated during the setup of Nessie for the purposes of the testing. Beyond these two points, we do not address the problem of generating complex objects in this work.

Adding new extension points. After executing a generated test, *extensionPoints* is called to identify where to extend the test in future iterations. This function returns an extension point for each callback node that has been executed, corresponding to the insertion of a new child node to the right of its existing children (see Figure 2b). There are two reasons for *not* adding 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. A single test may have multiple extension points because more than one of its callbacks may execute. Note that extension points are not removed after being used, so the set of extension points grows monotonically, enabling for multiple new tests to be derived from an extension point in a single base test.

4.3 Mining API Usages

Having an API call nested in the callback argument of another API call implies a relationship between these calls. We define a notion of a *nesting example* to formalize such relationships.

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} (i.e., a sync or async callback argument) or _ (any 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* (i.e., the same argument as given to f_{outer} at position k), *cb@k* (i.e., the kth parameter of the callback function), or _ (i.e., any other argument).

Example 4.1. The following nesting example can be mined from Figure 2a: $(\mathit{ensureFile}, (\_, \_, cb_{async}), \mathit{readJson}, (\mathit{outer}@0))$.

Example 4.2. The following usage of the `fs-extra` API, where the `obj` parameter of the callback passed to `readJson` serves as an argument in a nested call to `outputJson`:

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

corresponds to: $(\mathit{readJson}, (\_, cb_{async}), \mathit{outputJson}, (cb@1, \_))$.

We developed a static analysis for mining nesting examples from real-world uses of APIs by traversing the ASTs of existing API clients. This analysis was implemented in CodeQL [15], using its extensive facilities for static analysis. In particular, we use CodeQL's access path tracking to identify functions as originating from an API import, and its single static assignment representation of local variables to identify situations where the same argument is used in an outer call and an inner call. For shared arguments that are primitive values (e.g., the same string passed to both inner and outer calls) the relationship is identified by checking for value equality.

The test generator uses the set of mined nesting examples in *chooseFunction*. When selecting a function to be nested in the callback of some function f, the set of nesting examples is consulted to find examples where f_{outer} matches f. If such nesting examples exist, then one of the corresponding f_{inner} functions is randomly selected to be invoked inside f's callback. Similarly, *chooseArgument* consults the selected nesting example to determine which arguments (if any) to reuse from the outer function or from the surrounding callback, and at what position(s).

If no relevant mined nesting examples are available, an inner function is randomly selected. The test generator is configured to only use mined data 50% of the time. If we only used nestings that showed up in mined data, this would exclude many potentially correct pairs that simply do not occur in the mined projects. Section 5 explores the effect of varying how often mined data is consulted on the coverage achieved by the tests.

5 EVALUATION

Our evaluation aims to answer the following research questions:

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

Benchmarks. Table 1 shows the libraries we use as benchmarks for our evaluation, along with the number of lines of code (LOC) and the commit of the version we use. To select candidate libraries, we first identified two domains of libraries that commonly have asynchronous functionality: file systems and promises. We then picked popular libraries in those domains that satisfied the base requirements of: successfully installing/building, and having a test suite, with tests that all pass[5]. As a point of reference, we measure the statement coverage of the library code achieved by simply loading the library (column "Cov. loading" in Table 1). 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.

Table 1: Summary of projects used for evaluation.

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

Table 2: Abstract signatures categorized manually (M) and found by automated API discovery (A).

	Signatures *with* callbacks				Signatures *without* callbacks			
Project	**M**	**A**	**Only M**	**Only A**	**M**	**A**	**Only M**	**Only A**
`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`	–	15	N/A	N/A	–	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

Baselines and variants of the approach. We compare Nessie against the state of the art approach LambdaTester [35] (LT). Because the original LT does 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, we evaluate two variants of Nessie: *NES (seq)*, which uses sequencing only, and *NES (seq+nest)*, which uses both sequencing and nesting.

5.1 RQ1: Effectiveness of Automated API Discovery

To measure Nessie's effectiveness 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 3, if the approach does not discover any valid signatures for an API function then a default "callbackless" signature is assigned. We do not count this default signature towards the total number of discovered signatures.

[5] To automate the process of determining which libraries satisfy these requirements, we used npm-filter [8].

Table 2 displays the results of this experiment. For each API, we include the number of signatures found through manual documentation inspection and the number found through automated discovery, both with and without callback arguments. 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 automated discovery yields 88 signatures with callback arguments. Of those manually found, 3 are not discovered automatically; of those automatically discovered, 70 are not found manually. 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 does not provide function-level documentation.

We can summarize the effectiveness of the discovery phase by computing the percentage of documented signatures that are found with the automated approach. For the signatures with callback arguments, we compute this as follows:

$$1 - \frac{\text{signatures only found manually}}{\text{total number of signatures found manually}}$$
$$= 1 - \frac{3+4+7+7+33}{21+4+9+25+16+3+6+58} = \boxed{62\%}$$

With a similar computation, we see that the automated discovery finds 38% of the documented signatures without callback arguments.

Signatures found only manually. Nessie sometimes misses signatures because the automated discovery may fail to generate valid arguments, particularly in cases where 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 (e.g., `writeFileSync` and `readFileSync` in `jsonfile`). Another reason for signatures missed by the API discovery are functions that take *multiple* callback 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 the manual inspection of documentation. First, some functions are undocumented, e.g., 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 extra signatures. For example, consider the `writeFile` function in `jsonfile`. The documentation presents 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. Even though checking exposed APIs against documentation is not the primary purpose of Nessie, we made pull requests addressing this issue on both `fs-extra`[6] and `jsonfile`[7].

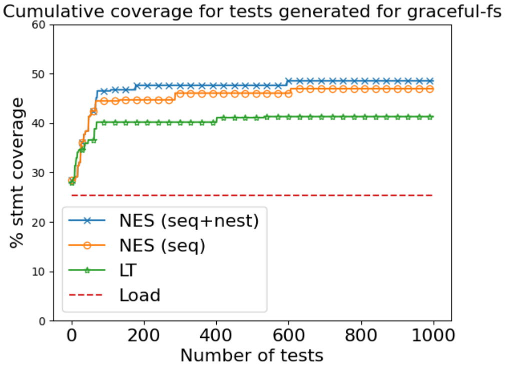


Figure 3: Cumulative coverage while generating 1,000 tests for graceful-fs.

Answer to RQ1: Automated discovery finds 62% of documented signatures that expect callback arguments, and 38% of signatures without callback arguments. It also discovers some undocumented signatures, which in several cases reflect unexpected behavior.

5.2 RQ2: Coverage Achieved by Generated Tests

To measure Nessie's effectiveness 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 with the two variants of Nessie, NES (seq+nest) and NES (seq), and with the baseline LT approach.

Figure 3 shows how cumulative coverage evolves while generating 1,000 tests for one package, `graceful-fs`. As a reference, the horizontal line shows the coverage directly after loading the library. The coverage follows a 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 Nessie, but the combination of sequencing and nesting converges more quickly. Moreover, Nessie achieves higher final coverage than LT.

The number of functions for which Nessie can generate tests and for which LT cannot depends entirely on the API. Therefore, the coverage improvements are quite variable across the packages tested. We chose `graceful-fs` as the demonstrative example because it shows both the plateauing of coverage and the relatively small

[6] https://github.com/jprichardson/node-fs-extra/pull/866
[7] https://github.com/jprichardson/node-jsonfile/pull/146

difference we generally see between Nessie with both nesting/sequencing and Nessie just sequencing. The same plot for all other packages are in the supplementary material.

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 Nessie to *match* and *exceed* the coverage that LT achieves after 1,000 tests. The last column shows the number of tests LT requires to reach the coverage it keeps at 1000 tests (i.e., the beginning of the coverage plateau LT reaches at 1000 tests). For example, for the `fs-extra` project, after 1,000 tests Nessie achieves a statement coverage of 37.2% with NES (seq+nest) and 35.4% with NES (seq), while LT achieves 34.4%. NES (seq+nest) matches and also exceeds LT's coverage after only 311 tests; NES (seq) matches and also exceeds LT's coverage after 663 tests. Meanwhile, after 889 tests, LT reaches a coverage plateau that it keeps until 1000 tests.

Overall, Nessie 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 than LT requires to reach the same coverage. Comparing the two variants of Nessie, the combination of sequencing and nesting is more effective.

Answer to RQ2: Nessie 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.

5.3 RQ3: Finding Behavioral Differences during Regression Testing

To answer RQ3, we use the tests generated by Nessie to find behavioral differences in consecutive commits of the benchmark libraries.

5.3.1 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 `mocha` testing framework [2] to run our tests. This framework can internally handle errors in a test, so that the remaining tests are executed even if an error is thrown in one of them. Most relevant for us, `mocha` handles errors thrown by asynchronously invoked functions and logs this as an *internal async error*. To detect behavioral differences, Nessie 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 call differs between commits.
- An argument to an API 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.

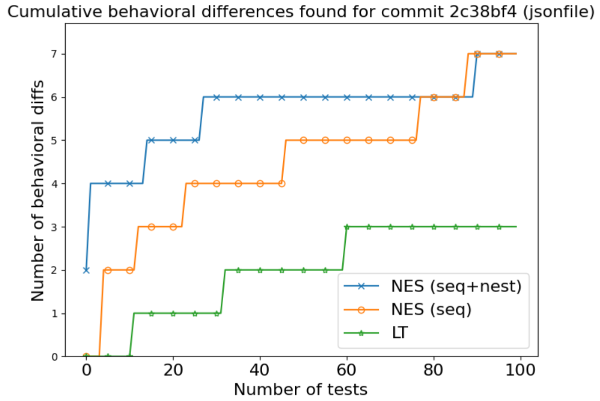


Figure 4: Cumulative number of behavioral diffs, for specific commit of `jsonfile`.

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.

5.3.2 Quantitative Results. As a representative example, Figure 4 shows the cumulative number of detected behavioral differences while generating 100 tests for a representative commit of `jsonfile`. As can be seen in the figure, NES (seq+nest) is most effective, followed by NES (seq), and then the existing LT technique.

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 table's first row reads: For `fs-extra`, we run the regression testing over 99 pairs of commits. NES (seq+nest) spots a behavioral difference in 17 of these pairs, with 2.8 unique differences per pair, on average. NES (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: Nessie finds more behavioral differences than LT

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	**NES (seq+nest)**	**NES (seq)**	**LT**	**NES (seq+nest)**	**NES (seq)**	**LT**
`fs-extra`	37.2%	35.4%	34.4%	311 (311)	663 (663)	889
`jsonfile`	87.2%	80.9%	80.9%	107 (209)	95 (N/A)	117
`node-dir`	32.5%	32.5%	32.5%	96 (N/A)	115 (N/A)	199
`bluebird`	48.0%	47.1%	47.1%	433 (433)	644 (N/A)	644
`q`	67.2%	66.4%	67.1%	103 (105)	765 (765)	980
`graceful-fs`	48.5%	47.0%	41.3%	49 (53)	49 (53)	534
`rsvp.js`	66.0%	66.0%	64.8%	177 (268)	196 (268)	325
`glob`	36.1%	36.1%	36.1%	3 (25)	181 (N/A)	76
`zip-a-folder`	32.0%	24.0%	24.0%	1 (793)	3 (N/A)	3
`memfs`	55.5%	48.3%	48.3%	84 (84)	702 (N/A)	702

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

		With diff. (avg. unique per diff)		
Project	**Compared pairs**	**NES (seq+nest)**	**NES (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)

(166 vs. 92 in total), and the combination of nesting and sequencing is worthwhile (166 vs. 135 differences in total).

5.3.3 Qualitative Results.

To better understand the behavioral differences that Nessie reveals, we manually inspect a random sample of them. Table 5 summarizes the results. For each analyzed difference, 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.
- **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 Nessie detects a variety of different types of API functionality changes. Interestingly, for several commits that introduce a bug, Nessie later finds the "dual" commit that fixes that same bug.

To better understand the impact of nesting, we checked for each inspected behavioral difference whether it could also be exposed by a test case without nesting. Similar to the example mentioned in the introduction, we find that creating a sequence-only test case is, in principle, possible for each of the behavioral differences. The key benefit provided by nesting is to more quickly cover a diverse set of behaviors than with purely sequential tests, and hence, to expose more behavioral differences in a given testing budget.

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
`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)
`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
`eaab0ee`	`glob`	Upgrade	`glob` succeeds in newer commit (error in older commit)
`5d8a060`	`zip-a-folder`	Upgrade	`zipFolder` executes callback in newer commit

Answer to RQ3: Nessie 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.

5.4 RQ4: Impact of Guidance by Mined Nesting Examples

The API usage mining component of Nessie 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.

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%

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 table's first row reads as follows: For `fs-extra`, the statement coverage when using 0% mined nestings is 32.2%, when using 25% mined nestings it is 33.6%, when using 50% mined nestings it is 37.2%, when using 75% mined nestings it is 33.0%, and when using 100% mined nestings it is 33.0%. 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 a coverage that is higher than or at least as high as 0% and 100% mined nesting. This supports our initial hypothesis that choosing informed nestings is more likely to produce valid tests that will increase coverage. However, choosing only mined nestings, i.e, 100%, risks to miss valid nestings that are simply never seen during the API usage mining.

We currently do not have results on the impact of mined data use on the regression tests. However, the increase in coverage caused by mining API usages, which we observe for 6 of the 10 libraries, suggests that mining may also help during regression testing.

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.

5.5 RQ5: Performance of Test Generation

We measure the time taken to generate 100 tests for each of three approaches we consider, including the discovery phase. 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 Nessie is implemented in TypeScript, it is single-threaded and so only uses one core. Depending on the library, the test generation takes between 15 and 30 seconds. The results show that the time required to generate 100 tests is fairly similar across all three approaches.

Mining the nesting examples is an up-front cost. As described in Section 4.3, we wrote a static analysis in CodeQL [15] to identify nesting examples; we ran this over a set of 13,580 JavaScript projects on GitHub using npm-filter [8]. This process took around 20 hours. This set of examples comes with the tool, and users of the tool would only be required to rerun the mining of nesting examples if they wanted to generate tests for APIs for which we did not mine any nesting examples. The cost of mining nesting examples depends only on the number of mined projects, and not on the number of APIs being mined for.

Answer to RQ5: With 15 to 30 seconds per 100 tests, the approach is efficient enough for practical use. Extending the set of mined nesting examples takes time proportional to the number of projects mined but is an up-front cost.

5.6 Threats to Validity

Internal validity. Our results may be influenced by several factors. First, our baseline is a re-implementation of LT [35] because the original implementation was not functional on our benchmarks, and their benchmarks do not contain asynchronous APIs. The re-implementation is based on the original code, which is publicly available, and we clarified questions on the code with the LT authors. Second, our automated identification of equivalent behavioral differences is approximate and may both over- and underapproximate a (theoretical) precise approach. Since the approximation is likely the same for all evaluated approaches, it should not influence the overall conclusions. Finally, the results of manually inspecting and classifying behavioral differences is subject to our understanding of the tested libraries. To mitigate this threat, we discussed all cases among the authors.

External validity. The libraries used in the evaluation may not be representative for the overall population of libraries with asynchronous callbacks. As of July 2021, `npmjs.com` reports a total of 116,342 packages that are dependent on the packages used in the evaluation, i.e., the benchmarks at least cover a relevant subset of all libraries. Finally, our implementation targets JavaScript, and hence, we do not claim our empirical results to generalize to other languages. Because callbacks, both synchronous and asynchronous, exist in various other languages, where they cause similar challenges for test generators, we hope our conceptual contributions may inspire future work for other languages.

6 RELATED WORK

Test generation for higher-order functions. There are several techniques for automatically testing higher-order functions. Most closely related is LambdaTester [35], which also targets JavaScript and has inspired some of our design decisions. The main difference is that Nessie 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. Other test generators can be roughly categorized into random testing and solver-based, systematic testing. QuickCheck [11] is an example of the former, as it creates callback functions that return a random, type-correct value. Koopman and Plasmeijer [20] 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 [19], 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 [27, 44], and work that performs a type-directed, enumerative search over the space of test cases [37]. Palka et al. [31] 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.

Random test generation. Nessie builds upon a rich history of random test generators, starting with Randoop [30], which introduced feedback-directed random test generation. Our work also follows this paradigm, but in contrast to Randoop, addresses challenges of higher-order functions and those arising in a dynamically typed language. EvoSuite [13] uses an evolutionary algorithm to continuously improve randomly generated test cases. Beyond function-level testing, application-level fuzzing has received significant attention, including AFL[8] and its derivatives [9, 22], and combinations of fuzzing with symbolic testing [39]. In contrast to the above greybox or whitebox fuzzers, Nessie does not need to analyze the library under test, but obtains feedback from the execution of the generated tests and, optionally, uses existing API clients for guidance.

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 [14]. These results show that generating tests without considering asynchronous callbacks (i.e., the tests that prior work [35] 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 [41]. 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 [43], suggesting that the problem we address is relevant beyond JavaScript.

Alimadadi et al. [5] propose a dynamic analysis to trace and visualize JavaScript executions, with a focus on asynchronous interactions across the client and the server. Another dynamic analysis detects promise-related anti-patterns [6]. Several techniques aim at detecting races in JavaScript [3, 4, 32, 34, 45], 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 Nessie.

There are several formalizations of different aspects of asynchronous JavaScript, including an execution model of Node.js [23], a model to reason about promises [24], and a calculus, semantics, and implementation of a static analysis of asynchronous behavior [38]. 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.

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 [7]. Examples of techniques include dynamic analyses to detect type inconsistencies [33], to detect inefficient code [16], to detect various common programming mistakes [17], and to reason about taint flows [18]. Work on reasoning about API changes and how they affect clients [26] is a recent example of a static analysis. Similar to Nessie, all these analyses take a pragmatic approach toward addressing the idiosyncrasies of JavaScript, without providing strong soundness or completeness guarantees.

7 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 in meaningful ways. Generating only sequences of calls, as done in existing test generators, is inadequate, as it is difficult for such an approach to produce suitable values to invoke API functions with.

We presented Nessie, the first test generator aimed at APIs with asynchronous callbacks, which relies on both sequencing and nesting API calls to produce suitable values to invoke API functions with. Here, nesting here means generated tests may contain API calls inside the body of callbacks passed to other API calls. In an empirical evaluation, Nessie is applied to ten popular JavaScript libraries containing 142 API functions with callbacks, and its effectiveness is compared to that of LambaTester, a state of the art test generation technique that creates tests only by sequencing method calls. Our results show that Nessie finds more behavioral differences and achieves slightly higher coverage than LambdaTester. Notably, it needs to generate significantly fewer tests to achieve and exceed the coverage achieved by LambdaTester.

TOOL/DATA AVAILABILITY

A full working artifact, including all experimental data associated with this research is available at: https://zenodo.org/record/5874851.

ACKNOWLEDGEMENTS

The authors would like to thank Rob Durst for his contributions to Nessie's discovery phase.

This work was supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and Perf4JS projects. E. Arteca and F. Tip were supported in part by National Science Foundation grants CCF-1715153 and CCF-1907727. E. Arteca was also supported in part by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] 2020. ECMAScript 2020 Language Specification. https://www.ecma-international.org/ecma-262/11.0/.
[2] 2021. Mocha. https://mochajs.org/. Accessed: 2021-08-24.
[3] 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

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

[4] 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

[5] 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.

[6] 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

[7] 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).

[8] Ellen Arteca and Alexi Turcotte. 2022. npm-filter: Automating the mining of dynamic information from npm packages. (2022). https://arxiv.org/abs/2201.08452

[9] 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

[10] 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.

[11] 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

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

[13] 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.

[14] 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.

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

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

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

[18] 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

[19] 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

[20] 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

[21] Erik Krogh Kristensen and Anders Møller. 2017. Type test scripts for TypeScript testing. *PACMPL* 1, OOPSLA (2017), 90:1–90:25.

[22] 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.

[23] 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

[24] 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

[25] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 304–315. https://doi.org/10.1109/ICSE.2019.00045

[26] 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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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

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

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

[34] 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)*.

[35] 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.

[36] 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.

[37] 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

[38] 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

[39] 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.*

[40] 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.

[41] 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.

[42] 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.

[43] 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

[44] 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

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