Test Generation for Higher-Order Functions in Dynamic Languages

MARIJA SELAKOVIC, TU Darmstadt, Germany
MICHAEL PRADEL, TU Darmstadt, Germany
REZWANA KARIM, California, USA
FRANK TIP, Northeastern University, USA

Test generation has proven to provide an effective way of identifying programming errors. Unfortunately, current test generation techniques are challenged by higher-order functions in dynamic languages, such as JavaScript functions that receive callbacks. In particular, existing test generators suffer from the unavailability of statically known type signatures, do not provide functions or provide only trivial functions as inputs, and ignore callbacks triggered by the code under test. This paper presents *LambdaTester*, a novel test generator that addresses the specific problems posed by higher-order functions in dynamic languages. The approach automatically infers at what argument position a method under test expects a callback, generates and iteratively improves callback functions given as input to this method, and uses novel test oracles that check whether and how callback functions are invoked. We apply *LambdaTester* to test 43 higher-order functions taken from 13 popular JavaScript libraries. The approach detects unexpected behavior in 12 of the 13 libraries, many of which are missed by a state-of-the-art test generator.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; *Dynamic analysis*;

Additional Key Words and Phrases: Higher-order functions, differential testing, dynamic analysis, JavaScript

ACM Reference Format:
Marija Selakovic, Michael Pradel, Rezwana Karim, and Frank Tip. 2018. Test Generation for Higher-Order Functions in Dynamic Languages. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 161 (November 2018), 27 pages. https://doi.org/10.1145/3276531

1 INTRODUCTION

Testing is widely accepted as one of the most effective techniques for detecting bugs. To reduce the human effort required for testing, various test generation approaches have been proposed, including feedback-directed random testing [Pacheco and Ernst 2007; Pacheco et al. 2008], symbolic execution [Cadar et al. 2008; King 1976], concolic execution [Godefroid et al. 2005; Sen et al. 2005], bounded exhaustive testing [Boyapati et al. 2002], evolutionary test generation [Fraser and Arcuri 2011], UI-level test generation [Memon 2007; Mesbah and van Deursen 2009; Selakovic et al. 2017], and concurrency testing [Pradel and Gross 2012; Samak and Ramanathan 2014]. Many of these approaches have discovered previously unknown bugs in mature and well-tested software, demonstrating the usefulness of test generation.

Higher-order functions are a programming language feature that has not yet received much attention from the test generation community. Such functions receive as arguments other functions,

Authors' addresses: Marija Selakovic, TU Darmstadt, Germany, m.selakovic89@gmail.com; Michael Pradel, TU Darmstadt, Germany, michael@binaervarianz.de; Rezwana Karim, California, USA, rezwanak@acm.org; Frank Tip, Northeastern University, USA, f.tip@northeastern.edu.

2475-1421/2018/11-ART161
https://doi.org/10.1145/3276531

which are later called back. Higher-order functions are particularly common for functional-style programming, e.g., using the popular `map` or `reduce` APIs, and in dynamic languages, e.g., to compose behavior via synchronous or asynchronous callbacks. For example, in JavaScript, the use of asynchronous callbacks is particularly common due to the single-threaded execution model of the language.

Despite the prevalence of higher-order functions, generating effective tests for them is a largely unsolved problem. Testing a higher-order function requires the construction of tests that invoke the function with values that include callback functions. To be effective, these callback functions must interact with the tested code, e.g., by manipulating the program's state. Existing test generators do either not address the problem of higher-order functions at all or pass very simple callback functions that do not implement any behavior or return random return values [Claessen and Hughes 2011].

The problem of generating higher-order functions is further compounded for dynamically typed languages, such as JavaScript, Python, or Ruby. For these languages, in addition to the problem of creating an effective callback function, a test generator faces the challenge of determining *where* to pass a function as an argument. Addressing this challenge is non-trivial in the absence of static type signatures.

This paper tackles the problem of automatically testing higher-order functions in dynamic languages by presenting a novel test generation framework called *LambdaTester*. In this framework, test generation proceeds in two phases. The *discovery phase* is concerned with discovering, for a given method[1] under test *m*, at which argument position(s) the method expects a callback function. To this end, the framework generates tests that invoke *m* with callback functions that report whether or not they are invoked. Then, the *test generation phase* creates tests that consist of a sequence of calls that invoke *m* with randomly selected values, including function values at argument positions where the previous phase discovered that functions are expected. Both phases take as input setup code that creates a set of initial values, which are used as receivers and arguments in subsequently generated calls.

We present several instantiations of the *LambdaTester* framework that differ in the way in which callback functions are constructed during the test generation phase. These instantiations include the use of: (i) empty functions, (ii) functions that return random values [Claessen and Hughes 2011], (iii) callbacks mined from a corpus of existing code, and (iv) a novel feedback-directed technique that generates callbacks using guidance from a dynamic analysis. Technique (iv) observes memory locations that are read during the execution of previously generated tests and generates function bodies that write to those locations.

We implement our ideas in a test generation tool for JavaScript. In an empirical evaluation, we use *LambdaTester* to generate tests for 43 higher-order functions in 13 popular JavaScript libraries. These libraries provide so-called *polyfills*, i.e., JavaScript implementations of APIs that may not be provided natively by all execution environments of the JavaScript language. We apply *LambdaTester* to polyfills for array APIs, including the *es5-shim*, *mozilla*, and *polyfill.io* libraries, and to polyfills of the promise APIs, including *bluebird*, *Q*, and *when*. To evaluate the effectiveness of the generated tests, we execute the tests both with the library implementation and the corresponding native implementation of the tested API, and detect situations where their behaviors differ. Here, behavioral differences are detected using an automated test oracle that compares execution behavior, e.g., the values being returned by the methods under test, the invocations of callback functions passed as arguments, the output written by the tested code, and whether the methods under test terminate.

[1] We use the terms "function" and "method" interchangeably in this paper because our approach tests methods while the term "higher-order function" is well established.

Our experimental results show that *LambdaTester* reveals various behavioral differences between polyfills and their corresponding native implementations, including previously unknown bugs in popular polyfills. Overall, the approach detects differences in 12 of 13 libraries. Comparing the different techniques for creating callback functions shows that callbacks that modify program state in non-obvious ways are more effective than simpler approaches. The most effective technique for creating callbacks is our novel feedback-directed technique, exposing differences missed by all other techniques.

The problem of testing higher-order functions is orthogonal to other challenges in test generation. We believe that by exploring different approaches to address this problem, our work not only provides a novel technique by itself, but will also provides guidance on tackling this problem in other test generators.

The remainder of this paper is organized as follows. Section 2 presents several motivating examples that illustrate the challenges associated with generating tests for higher-order functions. Section 3 presents our test generation approach. Section 4 discusses the test oracles used the evaluate the effectiveness of our approach, including novel callback-related oracles. The implementation of *LambdaTester* is discussed in Section 5. An evaluation of our approach is presented in Section 6. Finally, we discuss related work in Section 7 and conclude in Section 8.

2 CHALLENGES AND MOTIVATING EXAMPLES

This sections presents and illustrates challenges associated with generating effective tests for programs with higher-order functions in the context of dynamically typed languages. Given one or more methods under test *m* that expect a callback function *cb* as an argument, we identify five challenges for testing *m*:

(C1) Determining where *m* expects a callback function as an argument.
(C2) Generating callback functions *cb* that modify memory locations in such a way that it influences the behavior of *m*.
(C3) Generating callback functions *cb* that return values that influence the behavior of *m*, or that modify properties of objects passed into *m* as the receiver or as arguments.
(C4) Generating tests that chain multiple calls to higher-order functions.
(C5) Detecting callback-related behavioral differences during the execution of *m*.

The above challenges are relevant for any code that uses higher-order functions in a dynamically typed language. We now illustrate these challenges using two examples. Both examples are concerned with generating tests that expose bugs in *polyfills* for JavaScript, i.e., code that implements a feature that is unavailable in cases where a user is running an application using an outdated version of a browser or JavaScript engine.

2.1 Array.prototype.map

Figure 1 shows the implementation of `Array.prototype.map` from *polyfill.io* version 3.25.1. This code provides an implementation of the `map` method for use on platforms that predate JavaScript 1.6, where the method was introduced. A brief review of the code reveals that lines 2–4 check that the receiver is an object and throw a `TypeError` otherwise, and that lines 5–7 check that the callback is a function and throw a `TypeError` otherwise. On lines 9–12, several variables are initialized. If the function is invoked on an array, the variables `arraylike` and `length` contain the array and the array's length, respectively. If the receiver object is a string value, then variable `arraylike` is initialized to an array of which the elements contain the string's characters. Variable `result` is initialized to an empty array. The loop on lines 14–18 visits each index in the original array, looks

```
 1 Array.prototype.map = function map(callback) {
 2   if (this === undefined || this === null) {
 3     throw new TypeError(this + ' is not an object');
 4   }
 5   if (typeof callback !== 'function') {
 6     throw new TypeError(callback + ' is not a function');
 7   }
 8
 9   var object = Object(this), scope = arguments[1],
10       arraylike = object instanceof String ? object.split('') : object,
11       length = Math.max(Math.min(arraylike.length, 9007199254740991), 0) || 0,
12       index = -1, result = [];
13
14   while (++index < length) {
15     if (index in arraylike) {
16       result[index] = callback.call(scope, arraylike[index], index, object);
17     }
18   }
19
20   return result;
21 };
```

Fig. 1. Implementation of `Array.prototype.map` from polyfill.io.

```
22 p = ["a", "b", "c"];
23 q1 = p.map(function(v){ return v+v; });
24 q2 = p.map(function(v){ p.length = false; return v+v; });
```

Fig. 2. Examples of map method.

up the value at that index, computes a new value by invoking the callback function, and stores the result at the corresponding index in the `result` array.

The code in Figure 1 computes the expected results on most but not all inputs. For example, consider the call to `map` on line 23 in Figure 2. The function in Figure 1 assigns to `q1` an array `["aa", "bb", "cc"]` as expected. However, the result of the second `map` call on line 24 is equal to `["aa"]`, whereas the native implementation of `Array.prototype.map` assigns to `q2` an array `["aa", undefined, undefined]`.

Detecting this behavioral difference requires a test that passes a callback function as the first argument (see C1) and this callback function should manipulate the `length` property of the array `p` on which the `map` method is invoked (see C2). While existing test generators for JavaScript are able to generate simple callback functions, we are not aware of a previous technique that generates callback functions that modify specific properties of objects passed in as arguments, which is necessary to expose the bug in the `map` method in Figure 1. In this paper, we explore a technique that identifies object properties, such as `arraylike.length`, that are read in the method under test, and that generates callbacks that deliberately manipulate these properties.

2.2 Promises

Promises are a mechanism for asynchronous programming that was introduced in the ECMAScript 6 specification. A promise represents the value of an asynchronous computation, and it is in one of three states: pending, fulfilled, or rejected. Initially, a promise is in the pending state, and it

```
var p0 = new Promise(function(resolve,reject){ resolve(undefined) });
var p1 = p0.then(function(v){ return p1; });
var p2 = p1.then(function(v){ console.log("Value: " + v); });
var p3 = p2.catch(function(e){ console.log("Error: " + e); });
```

(a) Example of the circular promise chain.

```
var p0 = new Promise(function(resolve,reject){ resolve(7) });
var p1 = p0.then(undefined);
var p2 = p1.then(function(v){
  console.log(v);
  return v+1;
});
```

(b) Repeated calls to then.

```
var p = Promise.reject(17);
p.catch(function (){ console.log("hello"); },null,false);
```

(c) Call to catch with multiple arguments.

Fig. 3. Examples of promise calls.

transitions to the fulfilled or rejected state when functions `resolve` or `reject` are invoked, passing a value as an argument. To enable programmers to associate *reactions* with a promise, promises define higher-order functions `then` and `catch`, which receive callback functions that execute asynchronously when that promise is resolved or rejected. These operations enable programmers to create a chain of asynchronous computations and propagate errors from one asynchronously executed function to the next.

At the time of writing this paper, a popular web site[2] lists 76 polyfill implementations of JavaScript promises that aim at conforming to the Promises/A+ specification[3] upon which the ECMAScript 6 specification is based. Testing these implementations is a challenging task for several reasons:

- `then` and `catch` can be invoked with arguments that are functions but also with non-function values,
- `then` and `catch` return another promise, thus enabling programmers to create a chain of asynchronous computations, and
- `then` can be invoked with one argument to define a fulfill reaction, or with two arguments, to define both a fulfill reaction and a reject reaction, and
- the behavior of reactions defined using `then` and `catch` depends on the fact whether or not the returned value is a promise.

As we discuss in Section 6, our test generation technique finds numerous test cases that expose situations where polyfill implementations behave differently from the native implementation. For example, consider the example in Figure 3a. When this test is executed using Node.js 8.5.0, i.e., the native implementation, then it prints:

```
Error: TypeError: Chaining cycle detected for promise #<Promise>
```

The *bluebird* promise polyfill[4] prints a slightly different message:

```
Error: TypeError: circular promise resolution chain
```

[2] https://promisesaplus.com/implementations

[3] http://wiki.commonjs.org/wiki/Promises/A

[4] https://github.com/petkaantonov/bluebird.

These messages allude to the fact that the value being returned by the callback function on line 26 is the same value that is being returned by the call to `then` on the same line. However, the *Q* promise polyfill[5] fails to perform a circularity check and goes into an infinite loop that never terminates, which is clearly undesired behavior. Exposing this bug in *Q* requires generating a function that returns the same promise `p1` that is returned by the first call to `then` at line 26 (see C3). We are unaware of previous test generation techniques for JavaScript that are capable of generating such tests.

The example in Figure 3b illustrates some of the other complexities that arise in generating effective tests in the presence of higher-order functions. Here, a chain of promises is constructed using repeated calls to `then`. Note that, on line 30, the value `undefined` is passed to `then` instead of a function value. According to the specification, this should be equivalent to passing the identity function `function(v){ return v; }`, and executing the code using the native promises implementation prints "`7`". However, the polyfills provided by *Q* and *When*[6] print "`[Function]`" instead. Note that, to expose these errors, it was necessary to generate a test that contains a chain of function calls (see C4), and that it requires the test generator to generate calls to `then` with arguments that are both functions and non-function values.

As a final example, consider the test case in Figure 3c. In this example, the native implementation of promises executes without any errors and prints "`hello`". However, *bluebird* throws an uncaught exception

```
Unhandled rejection TypeError: Cannot read property 'apply'of null
```

without printing any output. Further investigation reveals that, in this case, the callback is not invoked by *bluebird* (see C5).

3 FRAMEWORK FOR TESTING HIGHER-ORDER FUNCTIONS

This section presents our *LambdaTester* framework for testing higher-order functions. Given a set of methods under test and, optionally, some setup code required to test these methods, the framework generates tests that invoke the methods under test. The key novelty of *LambdaTester* is to effectively test methods that receive other functions, i.e., callbacks, as arguments. To support testing of such higher-order functions, the framework consists of two phases. The first phase, called *discovery phase*, infers for each method under test at what argument positions the method expects a callback argument (Section 3.1). The second phase, called *test generation phase*, creates tests that pass callback functions and other argument values to the methods under test (Section 3.2). The test generation phase uses a form of feedback-directed, random testing [Pacheco and Ernst 2007] to incrementally extend and execute tests. We augment feedback-directed, random testing with four techniques to create callback arguments.

Before presenting the details of *LambdaTester*, we define our terminology. Each tests begins with a piece of user-provided setup code:

Definition 3.1 (Setup code). Setup code *setup* is a sequence of pairs (var, exp), where *var* is a variable name and *exp* is the expression assigned to *var*.

The purpose of the setup code is to create a set of values to be used as receivers or arguments of method calls. For example, to test methods on promises, the user needs to provide setup code that creates some initial promise objects. For the test in Figure 3a, the first line shows the setup code of the test.

The basic ingredient of generated tests are method calls:

[5] See https://github.com/kriskowal/q.

[6] See https://github.com/cujojs/when.

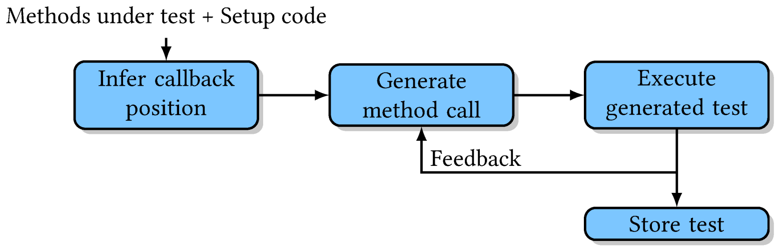

Fig. 4. Overview of the approach.

Definition 3.2 (Method call). A method call c is a tuple $(m, var_{rec}, var_{arg1} \cdots var_{argn}, var_{return})$, where m is a method name, var_{rec} is the name of the variable used as the receiver object of the call, $var_{arg1}, ..., var_{argk}$ are the names of variables used as arguments and var_{return} is the name of the variable to which the call's return value is assigned.

Finally, the overall goal of the approach is to generate tests:

Definition 3.3 (Test). A test *test* is a sequence $(setup, c_i, ..., c_n)$ where *setup* is the setup code and $c_i, ..., c_n$ are generated method calls.

Figure 4 illustrates the process of test generation. For each method under test, the approach attempts to infer the positions of callback arguments. Afterwards, the approach repeatedly generates new method calls and executes the growing test. During each test execution, the approach collects feedback that guides the generation of the next method call. Finally, the approach stores the generated tests, which can then be used for bug finding (Section 4).

3.1 Discovery Phase: Inferring Callback Positions

In dynamic languages, the expected number and types of method parameters are generally unknown. In particular, our test generator cannot rely on static type signatures to decide where to pass a callback argument. To find out at which argument positions a method under test expects a callback, the *discovery phase* of our approach explores all possible callback positions. To this end, the approach creates tests that pass callbacks at each argument position, while leaving the number and types of the other arguments unconstrained. The approach then collects feedback from executing these tests to determine which callbacks are executed, allowing the approach to infer the argument positions where callbacks are expected.

Algorithm 1 illustrates our technique for finding callback positions for a given set M of methods and setup code *setup*. The output of the algorithm is a map C that maps each method name to a set of possible callback positions. As receiver objects and arguments of method calls the algorithm considers two sets of variables. First, we use a set V_{rand} of variables that store randomly generated values. To initialize this set, *LambdaTester* randomly generates values for primitive types, such as *strings*, *booleans*, and *numbers*, as well as common object types, such as *arrays* and *objects*. Moreover, we add *null* and *undefined* to the V_{rand} set. Second, we use the set V_{setup} of variables assigned to in the setup code. To obtain this set, the approach statically analyzes the setup code and extracts all declared variables. For example, after parsing the setup code in the first line of Figure 3a, V_{setup} contains the variable `p0`.

The main loop of the algorithm repeatedly invokes each method under test until exceeding the testing budget, e.g., a fixed number of method invocations. For each method call, the algorithm passes *max_params* arguments such that callback functions are passed at different argument positions. In our experiments, we set *max_params* to five because the higher-order functions we analyze do not expect more than five arguments. In many dynamic languages, such as JavaScript,

Algorithm 1 Algorithm to infer callback position

```
Input: Set $M$ of names of methods under test, setup code $setup$
Output: Map $C$ that maps each method name to a set of $P_{cb}$ of callback positions
Initialize $C[m]$ with an empty set for each $m \in M$
$V \leftarrow V_{setup} \cup V_{rand}$
for each $m \in M$ do
    while testing budget not exceeded do
        $test \leftarrow$ new test starting with $setup$
        for each $pos_{cb} < max\_params$ do
            $var_{rec} \leftarrow selectReceiver(V, m)$
            $args \leftarrow$ empty sequence
            for each $pos < max\_params$ do
                if $pos = pos_{cb}$ then
                    $var_{cb} \leftarrow$ callback function that logs calls to it
                    Append $var_{cb}$ to $args$
                else
                    $var_{arg} \leftarrow randomChoice(V)$
                    Append $var_{arg}$ to $args$
            Append $(m, var_{rec}, args, \_)$ to $test$
            $feedback \leftarrow execute(test)$
            if $feedback$ has non-empty log then
                Add $pos_{cb}$ to $C[m]$
return $C$
```

if a method is called with more arguments than expected, the redundant arguments are simply ignored. For every argument position pos_{cb}, the algorithm creates a new method call that passes a callback function as the argument at position pos_{cb} and randomly selected values from V at all other argument positions.

When creating a call to a method m, the algorithm selects the receiver object from those elements in V that have a property named m. This selection, indicated by *selectReceiver* in the algorithm, is based on feedback from executing the setup code and the code that initializes the values in V_{rand}. During this initial execution, the approach gathers type information about all variables in V, including which properties the values stored in these variables provide.

For example, to generate a call to the `catch` method of promises based on the setup code in the first line of Figure 3a, the algorithm may select `p0` as the receiver variable because it provides a method named `catch`. Likewise, to generate a call to the `reduce` method, the algorithm may select a receiver from a randomly generated array in V_{rand} because arrays provide a `reduce` property.

After preparing all variables involved in a method call, the algorithm creates and then executes a test that contains the setup code followed by the call. During the execution of the test, the algorithm gathers feedback on its execution. Specifically, the algorithm keeps track of whether the callback function passed at position pos_{cb} gets invoked. If the callback function gets invoked, the algorithm infers that a function argument is expected at this position and updates the map C accordingly.

Finally, after calling the methods under test with various different arguments, the algorithm returns the inferred callback positions.

```
function callback() {
};
```

(a) Empty callback (“Cb-Empty”).

```
function callback() {
  return null;
};
```

(b) Callback generated by QuickCheck (“Cb-QuickCheck”).

```
function callback() {
  return Math.floor(10.8) +
         Math.floor(20.4) +
         Math.min(3, 5);
};
```

(c) Callback mined from existing code (“Cb-Mined”).

```
function callback(a,b) {
  receiver.foo = "abc"
  b = null;
  return {x: 23};
};
```

(d) Callback generated based on dynamically analyzing the method under test (“Cb-Writes”).

Fig. 5. Examples of generated callbacks.

3.2 Test Generation Phase

After inferring at what argument positions the methods under test expect a callback argument, the second phase of *LambdaTester* creates tests that pass different kinds of callbacks to the methods under test. The approach combines feedback-directed, random test generation with different techniques for creating callback functions. In the following, we first present how *LambdaTester* creates callback functions (Section 3.2.1) and then describe the overall test generation algorithm (Section 3.2.2).

3.2.1 Generation of Callback Functions. Effectively testing higher-order functions requires callback functions to be passed as arguments to the methods under test. The *LambdaTester* framework currently supports four techniques for generating callback functions, which we present below.

Empty callbacks. The most simple approach for creating callbacks is to simply create an empty function that does not perform any computation and that does not explicitly return any value. Figure 5a gives an example of an empty callback. We consider this approach as a baseline for comparison with the following approaches.

Callbacks by QuickCheck. QuickCheck [Claessen and Hughes 2011] is a state-of-the-art test generator originally designed for functional languages. To test higher-order functions, QuickCheck is capable of generating functions that return random values, but the functions that it generates do not perform additional computations and do not modify the program state. There are several re-implementations of QuickCheck for dynamic languages. We integrated an implementation for JavaScript[7] into *LambdaTester*. Integrating other existing testing tools that generate callbacks into our framework would be straightforward.

Figure 5b gives an example of a callback generated by QuickCheck.

Existing callbacks. Given the huge amount of existing code written in popular languages, one way to obtain callback functions is to extract them from already written code. To find existing callbacks for a method *m*, the approach statically analyzes method calls in a corpus of code and extracts function expressions passed to methods with a name equal to *m*. For example, to test the `map` function of arrays in JavaScript, we search for callback functions given to `map`. The rationale for

[7] The supported implementation of QuickCheck is available at https://quickcheckjs.readme.io/

extracting callbacks specifically for a each method *m* is that callbacks for a specific API method may follow common usage patterns, which may be valuable for testing these API methods.

To extract existing callbacks, we consider JavaScript code provided by a popular code corpus [Raychev et al. 2016]. We analyze this code with an AST-based analysis that extracts all function expressions that are passed as an argument to a function. For each extracted function, the analysis stores the name of the called function along with the callback function. During test generation, *LambdaTester* then random selects from those extracted callbacks that match the current method under test. Figure 5c gives an example of an existing callback.

Callbacks generation based on dynamic analysis. The final and most sophisticated technique to create callbacks uses a dynamic analysis of the method under test to guide the construction of a suitable callback function. The technique is based on the observation that callbacks are more likely to be effective for testing when they interact with the tested code. To illustrate this observation, consider the following method under test:

```
function testMe(callbackFn, bar) {
  // code before calling the callback

  // calling the callback
  var ret = callbackFn();

  // code after calling the callback
  if (this.foo) { ... }
  if (bar) { ... }
  if (ret) { ... }
}
```

To effectively test this method, the callback function should interact with the code executed after invoking the callback. Specifically, the callback function should modify the values stored in `this.foo`, `ret`, and `bar`. The challenge is how to determine the memory locations that the callback should modify.

We address this challenge through a dynamic analysis of memory locations that the method under test reads after invoking the callback. We apply the analysis when executing tests, and feed the resulting set of memory locations back to the test generator to direct the generation of future callbacks. The basic idea behind the dynamic analysis is to collect all memory locations that (i) are read after the first invocation of the callback function and that (ii) are reachable from the callback body. The reachable memory locations include memory reachable from the receiver object and the arguments of the call to the method under test, the return value of the callback, and any globally reachable state. To gather the relevant memory locations, the dynamic analysis performs the following actions during the execution of the method under test:

- *Store arguments and receiver object at method entry.* When the execution of the method under test starts, the analysis stores the method arguments and the receiver object.
- *Track calls to callback function.* The analysis observes when the callback function is invoked and then starts to track memory reads.
- *Track callback-reachable variable reads.* When the analysis observes a read to a variable, it checks whether the variable is transitively reachable from the receiver object or the arguments of the call of the method under test, from the return value of the callback, or from the global object. If the value is reachable from one of these starting points, the analysis stores its access path to retrieve the value, i.e., a sequence of property accesses applied to the starting point object. For

example, consider the reads of `ret` and `bar` in the above example. Because both happen after the callback invocation and are memory locations reachable from the callback body, the analysis reports the access paths `ret` and `arg2`, where `arg2` refers to the second argument passed to the method under test.

- *Track callback-reachable property reads.* Similar to variable reads, the analysis checks for every property read whether the read value is reachable from the callback body. For example, consider the read of `this.foo` in the above example. As `this` refers to the receiver object of the call to `testMe`, the value can be reached via the *foo* property of the receiver object, i.e., the stored access path is `receiver.foo`. The callback function could modify this value before the read by writing to `receiver.foo`, where `receiver` is the variable in the test that refers to the receiver object.

For the above example, the set of dynamically detected memory locations is: { `receiver.foo`, `arg2`, `ret` }.

Based on the dynamically detected memory locations, *LambdaTester* generates a callback body that interacts with the function under test. To this end, the approach infers how many arguments a callback function receives by first executing the method under test with a callback that inspects *arguments.length*. Then, *LambdaTester* generates callback functions that write to the locations read by the method under test and that are reachable from the callback body. The approach randomly selects a subset of the received arguments and of the detected memory locations, and assigns a random value to each element in the subset.

Figure 5d shows a callback function generated for the above example, based on the assumption that the callback function receives two arguments. As illustrated by the example, the feedback from the dynamic analysis allows *LambdaTester* to generate callbacks that interact with the tested code by writing to memory locations that are relevant for the method under test.

3.2.2 *Feedback-Directed Test Generation.*

The callback functions generated by one of the four techniques presented in Section 3.2.1 are the core of *LambdaTester*. We now describe how the framework uses these callbacks and other values to generate tests in a feedback-directed, random manner. For methods under tests that expect function arguments according to the discovery phase of *LambdaTester*, the approach generates sequences of method calls that probabilistically pass callback arguments. For methods that do not expect callbacks, the approach generates sequences of calls with an unconstrained list of arguments.

Algorithm 2 illustrates our test generation approach. For a given set M of methods, the approach generates tests that contain sequences of calls to methods in M. The inputs to the algorithm are: (i) the set of methods M, (ii) user-provided setup code *setup*, and (iii) the map C, which maps each method name to a set of callback positions. The output of the algorithm is the set T of generated tests.

During test generation, the algorithm maintains two sets of values. First, it maintains the set V of variables, which – as for Algorithm 1 – comprises variables initialized in the setup code and in the generated tests, as well as randomly initialized variables. Second, the algorithm maintains a set R of memory locations, which are the output of the dynamic analysis of memory reads in the method under test. These locations serve as feedback that helps the test generator create effective callbacks (Section 3.2.1).

The algorithm incrementally generates method calls until a maximum number of calls *max_calls* per test is reached. To generate a method call, the algorithm randomly picks a name from M and a callback position *index* from C. The approach selects the receiver object from the variables V by randomly choosing only from elements in V that provide a method called *m*. For the callback argument, the algorithm invokes *generateCallback*, which implements one of the four techniques discussed in Section 3.2.1. Values of other non-function arguments are selected from the pool V.

Algorithm 2 Test generation algorithm

```
Input: Set $M$ of names of methods under test, setup code $setup$, map $C$
Output: Generated tests $T$
$T \leftarrow \emptyset$
$R \leftarrow \emptyset$
$V \leftarrow V_{setup} \cup V_{rand}$
while testing budget not exceeded do
    $test \leftarrow$ New test starting with $setup$
    while $max\_calls$ not reached do
        $m \leftarrow randomChoice(M)$
        $pos_{cb} \leftarrow randomChoice(C[m])$
        $var_{rec} \leftarrow selectReceiver(V, m)$
        $n \leftarrow randomChoiceInRange(pos_{cb}, max\_args)$
        $args \leftarrow$ empty sequence
        for each $pos \leq n$ do
            if $pos == pos_{cb}$ and $random() \leq use\_callback\_prob$ then
                $var_{arg} \leftarrow generateCallback(R, V)$
                Append $var_{arg}$ to $args$
            else
                $var_{arg} \leftarrow randomChoice(V)$
                Append $var_{arg}$ to $args$
            Create fresh variable $var_{ret}$ and add it to $V$
        Append $(m, var_{rec}, args, var_{ret})$ to $test$
        $feedback \leftarrow execute(test)$
        if $feedback$ indicates a crash then
            Add $test$ to $T$
            $break$
        Update $R$ with $feedback$
    Add $test$ to $T$
return $T$
```

The algorithm adds the variable var_{ret}, which stores the return value a newly added calls, to the set of variables V. That is, the test generator considers return values as a potential receiver objects or arguments in future calls.

After creating a method call, the algorithm adds the call to the current test, executes the test and collects feedback from the test's execution. The feedback consists of two kinds of information. First, the algorithm observes whether the generated test crashes by throwing an exception. In this case, further extending this test is not useful and the algorithm breaks out of the inner loop that appends further calls. Second, the algorithm receives feedback from the dynamic analysis of the memory locations read during the test execution and updates the set R by adding these locations to the set.

The main loop of the algorithm continues to create tests until the given testing budget has been exceeded.

4 TEST ORACLE: DIFFERENTIAL TESTING OF POLYFILLS

The primary goal of test generation techniques is to detect bugs. To assess the effectiveness of our testing framework in finding bugs we generate tests for polyfills that accept callback arguments.

In JavaScript parlance, a *polyfill* is a user-defined implementation of an API that provides a method's functionality in older execution environments that do not natively support it. For example, before ECMAScript 6 added native support for promises, the `Promise` object had been available in JavaScript through third-party libraries such as *bluebird* and *Q*. However, polyfills are non-trivial to implement because approximating all possible behaviors supported by the native implementation is sometimes very challenging. To find bugs in polyfills we use differential testing [McKeeman 1998] and consider the output of the native implementation as the ground truth.

When testing higher-order functions, it is often insufficient to determine whether the native implementation and the polyfill produce the same output state given the same input state. For example, two implementations can produce the same output for some inputs but invoke callback functions a different number of times, which clearly indicates that these implementations are not equivalent. In this section, we define test oracles relevant for testing higher-order functions, including novel callback-related oracles.

To compare test executions of native and polyfill implementations and find behavioral differences we define the notion of an execution summary.

Definition 4.1 (Execution summary). An *execution summary* captures the result of a test execution. It contains the following information:

- The state of receiver objects and return values of calls.
- The state of arguments passed to callback functions.
- The number of invocations of callback functions.
- Output written to the standard output stream.
- Output written to the standard error stream.

To record the state of an object in the execution summary, *LambdaTester* serializes the object. *LambdaTester* also serializes the arguments passed to callback functions for every callback invocation. For primitive types, serialization is straightforward: we simply store the values. When serializing objects not created by a constructor, e.g., arrays, we rely on the *JSON* serialization API for converting an object to its string representation. In contrast, if an object is created by a constructor (e.g., promises) the internal object representation depends on the constructor implementation, and this representation may vary across polyfills. In this case, we do not serialize the object, but record only the constructor name. Hence, if two promise libraries have the same behavior but use different representations for promise objects, then we consider them equivalent.

To compare two implementations, *LambdaTester* also considers output written to the standard output and standard error streams. Standard output contains the output produced by the test's execution, and standard error contains error messages thrown during the test's execution. A challenge when comparing output written to these streams is that different implementations of a polyfill tend to produce different warning messages and error messages. For example, *bluebird* and the native implementation produce different error messages when an attempt is made to construct a circular promise chain, as we discussed in Section 2.2. To avoid false positives due to different warning messages, we consider two implementations as different only if one implementation produces an empty message while the other produces a non-empty message.

Based on the definition of execution summary, *LambdaTester* considers the following eight oracles:

- Standard error - two implementations produce different standard error. Here we distinguish two sub-categories:
  - Error messages - indicates a situation when one implementation reports an error message and the other does not.

  - Warnings - similar to the previous sub-category but this compares warning messages.
- Non-termination - a situation where one implementation terminates and the other does not.
- Standard output - two implementations produce different standard output.
- State of receiver objects - differences in the state of receiver objects.
- State of return values - differences in the state of return values.
- Callback arguments - differences in the state of callback arguments.
- Callback invocations - differences exposed by the number of callback invocations.

The last two oracles are callback-related oracles. Our experimental evaluation (Section 6) shows that many behavioral differences would be missed if callback-related oracles were not considered.

5 IMPLEMENTATION

We implemented our approach in *LambdaTester*, a testing tool for JavaScript programs. The implementation is freely available for download.[8] The dynamic analysis is based on source-to-source instrumentation and builds on top of the dynamic analysis framework Jalangi [Sen et al. 2013]. We integrated the QuickCheck implementation available at https://quickcheckjs.readme.io.[9] This QuickCheck implementation provides interfaces for the generation of various JavaScript types, including functions, and we configure QuickCheck so that the functions that it generates may return a value of any type. Currently, this can be boolean, string, real, integer, array, object, and `undefined`. To mine existing callback arguments, we implement an AST-based analysis based on the acorn parser[10] and on estraverse[11]. Finally, to measure the coverage we use the Istanbul tool[12].

6 EVALUATION

We evaluate *LambdaTester* on 43 higher-order functions taken from 13 popular libraries. This section reports on experiments that aim to answer the following research questions:

RQ1: How effective are the different variants of *LambdaTester* in finding behavioral differences?
RQ2: What kinds of behavioral differences are detected by *LambdaTester*?
RQ3: How effective are different variants of *LambdaTester* in increasing code coverage?
RQ4: How efficient is *LambdaTester*?

6.1 Experimental Setup

Benchmarks. We evaluate our approach on higher-order functions taken from 13 popular JavaScript libraries listed in Table 1. Polyfill.io, mozilla, and es5-shim implement polyfills for eight array methods indicated in the table. The other ten libraries implement JavaScript promises. For the promise libraries, we select the most popular[13] implementations of promises that aim to be compatible with the ECMAScript 6 standard. To test promise polyfills, we consider two methods that expect callbacks as arguments: `then` and `catch`.

Test generation approaches. We compare the effectiveness of several variants of our testing approach as summarized in Table 2. The *Base* approach generates tests that call a single method with randomly selected arguments. It is unaware of callback arguments and never generates callbacks as arguments. The following four approaches correspond to the callback generation

[8] https://github.com/sola-da/LambdaTester

[9] Other JavaScript implementations of QuickCheck exist, e.g., JSVerify http://jsverify.github.io/, which could be integrated into *LambdaTester* as well.

[10] https://github.com/acornjs/acorn

[11] https://github.com/estools/estraverse

[12] https://istanbul.js.org/

[13] According to the star rating on github.com.

Table 1. Benchmarks used for the evaluation.

Name	Version	LoC	API methods
Polyfill.io	3.25.1	189	filter, find, every, some, forEach, map, reduce, reduceRight
Mozilla polyfills	-	199	filter, find, every, some, forEach, map, reduce, reduceRight
es5-shim	4.5.10	2098	filter, find, every, some, forEach, map, reduce, reduceRight
Q	1.5.1	1235	then, catch
bluebird	3.5.1	5188	then, catch
when	3.7.8	1844	then, catch
then/promise	8.0.1	567	then, catch
rsvp.js	4.8.2	963	then, catch
native-promise-only	0.8.1	292	then, catch
lie	3.3.0	309	then, catch
pacta	0.9.0	403	then, catch
es6-promises	1.0.10	274	then, catch
bloodhound-promises	1.4.14	652	then, catch

Table 2. Test generation approaches used for the evaluation.

Approach	Description	Feedback-directed	Infer callback positions	Callback functions
Base	Random test generator that is unaware of callbacks. It never passes any callback function as an argument.	No	No	—
Cb-Empty	A callback-aware test generator that infers arguments that expect callbacks, and that passes empty callback functions as arguments.	Yes	Yes	Empty functions
Cb-Quick	Like Cb-Empty but with callback functions generated by QuickCheck.	Yes	Yes	QuickCheck-generated functions
Cb-Mined	Like Cb-Empty but with callback functions mined from existing code.	Yes	Yes	Mined functions
Cb-Writes	Like Cb-Empty but with callback functions generated based on a dynamic analysis of memory reads.	Yes	Yes	Functions with targeted writes

techniques introduced in Section 3.2.1. The *Cb-Empty* approach infers which arguments expect callbacks, generates tests as sequences of function calls and passes empty callbacks to functions that expect them. *Cb-Mined* is like *Cb-Empty*, but with callback functions mined from existing code. *Cb-Quick* is also like *Cb-Empty*, but with callback functions generated by QuickCheck. Finally, *Cb-Writes* uses a dynamic analysis to determine relevant memory locations that are read by the method under test, and attempts to generate callbacks that write to those locations.

6.2 Effectiveness in Finding Behavioral Differences

Table 3 compares the different test generation approaches in terms of the number of behavioral differences exposed by 1,000 generated tests. The table includes the differences detected with test

Table 3. Comparison of different test generation approaches in terms of the number of behavioral differences exposed by 1,000 generated tests. We show results only for those *mozilla.js* and *polyfill.io* polyfills where at least one of the testing approaches finds a behavioral difference.

Benchmark	Base	Cb-Empty	Cb-Quick	Cb-Mined	Cb-Writes
Polyfill.io (map)	0	0	0	0	28
Polyfill.io (find)	0	0	0	0	19
Mozilla (filter)	0	0	0	7	36
es5-shim	0	0	0	0	0
Bluebird	0	475	423	264	409
Q	0	3	1	60	151
when	0	374	331	49	309
then/promise	0	0	0	57	122
rsvp.js	0	21	15	23	164
native-promise-only	0	41	30	100	184
lie	0	28	24	82	135
pacta	0	0	0	57	120
es6-promises	0	0	0	57	149
bloodhound-promises	0	63	51	153	183

oracles listed in Section 4, except for differences in the warning messages reported by libraries (as discussed in more detail in Section 6.3).

In total, we find behavioral differences in 12 out of 13 libraries. The *Base* approach does not find any difference in any polyfill. *Cb-Empty* and *Cb-Quick* find differences in 7 libraries, *Cb-Mined* in 11 libraries, and *Cb-Writes* in 12 libraries. Overall, the *Cb-Writes* approach outperforms all other approaches. The average number of differences found by *Cb-Writes* is the largest across all libraries. The largest number of differences per single library is found in *Bluebird*, which is perhaps surprising as it is the most popular promise implementation[14]. Furthermore, we consider differences in warning messages as benign differences and exclude them as errors in Table 3.

6.3 Classification of Behavioral Differences

We are aware that some of the behavioral differences we find are likely to be due to the same root cause. To better understand the types of behavioral differences, we classify each of them into one or more categories based on the oracles defined in Section 4. In the following, we show a breakdown of differences based on the way they manifest.

Tables 4 and 5 show how many behavioral differences are found in each category for each feedback-directed testing approach. Based on these results we can draw the following conclusions:

- Warnings, error messages, and differences in execution summaries are the dominant kinds of behavioral differences. Because the exact warning messages are not specified, these differences can likely be considered as false positives.
- For the promise libraries, several differences arise from different states of callback arguments, showing that considering callback-related oracles helps identify more behavioral differences. Whether a callback is called or not is an important behavioral property and polyfills should agree with the native implementation.
- Many promise libraries show equivalent behavior regarding their standard output and standard error. Because we test all libraries with the same generated tests, this is the reason for several identical numbers in the “Err” and “Warn” columns in Tables 4 and 5.

[14] According to the star rating on github.com.

Table 4. Behavioral differences found by *Cb-Empty* and *Cb-Quick* approaches in 1,000 generated tests per benchmark. Err = Error messages, Warn = Warning messages, NT = Non-termination errors, St.out = Standard output, Rec = Receiver objects, Ret = Return values, C.arg = Callback arguments, C.inv = Callback invocations.

Benchmark	Cb-Empty								Cb-Quick							
	Err	Warn	NT	St.out	Rec	Ret	C.arg	C.inv	Err	Warn	NT	St.out	Rec	Ret	C.arg	C.inv
Polyfill.io	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Mozilla	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
es5-shim	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bluebird	381	0	0	0	0	0	296	255	317	0	0	0	0	0	269	227
Q	0	394	0	0	0	0	3	0	0	377	0	0	0	0	1	0
when	297	69	0	0	0	0	177	141	245	67	0	0	0	0	171	137
then/promise	0	394	0	0	0	0	0	0	0	377	0	0	0	0	0	0
rsvp.js	0	394	0	0	0	0	21	0	0	377	0	0	0	0	15	0
native-promise-only	0	394	0	0	0	0	41	0	0	377	0	0	0	0	30	0
lie	0	394	0	0	0	0	28	0	0	377	0	0	0	0	24	0
pacta	0	394	0	0	0	0	0	0	0	377	0	0	0	0	0	0
es6-promises	0	394	0	0	0	0	0	0	0	377	0	0	0	0	0	0
bloodhound-promises	0	394	0	0	0	0	63	0	0	377	0	0	0	0	51	0

- Non-termination errors in promise libraries are detected by *Cb-Writes* only. The reason is that *Cb-Writes* uses objects created by previous method calls as possible return values for callbacks. This causes it to attempt to create circular promise chains, thus triggering non-termination errors. Since non-termination certainly is an undesirable property, the polyfills should not diverge from the native implementations w.r.t. this behavioral property.
- The *Cb-Writes* approach detects significantly more differences in array polyfills than other testing approaches. This result shows that using a more sophisticated approach for creating callbacks that interact with the method under test via shared state is worth the effort.

6.4 Array Polyfills Generated by Mimic

As another benchmark, in addition to the human-written libraries considered so far, we also apply *LambdaTester* to array polyfills generated by Mimic [Heule et al. 2015]. Mimic is a tool for synthesizing models for a variety of array-manipulating functions. The current implementation provides models for JavaScript's built-in array methods, including higher-order functions such as `filter`, `find`, `every`, `some`, `forEach`, `map`, `reduce` and `reduceRight`.

We evaluate the effectiveness of *LambdaTester* on all Mimic-synthesized array polyfills. Table 6 shows the number of behavioral differences found by each testing approach. Interestingly, all tests generated by *Base* approach show errors in mimic polyfills. This is because Mimic's polyfill implementations do not throw errors when a non-function argument is passed at positions where a callback is expected. Furthermore, all tests generated by *Cb-Empty* expose errors in Mimic's `every` and `some` polyfills. The reason is that these polyfills are supposed to return a boolean value, but when receiving an empty callback, they always return `undefined`.

In general, the polyfills generated by Mimic have significantly more differences from the native implementation than the human-written polyfill libraries. Since we are not aware of any use of synthesized mimic models in real-world applications, we exclude these polyfills as a point of comparison in Table 3.

Table 5. Behavioral differences found by *Cb-Mined* and *Cb-Writes* approaches in 1,000 generated tests per benchmark. Err = Error messages, Warn = Warning messages, NT = Non-termination errors, St.out = Standard output, Rec = Receiver objects, Ret = Return values, C.arg = Callback arguments, C.inv = Callback invocations.

Benchmark	Cb-Mined								Cb-Writes							
	Err	Warn	NT	St.out	Rec	Ret	C.arg	C.inv	Err	Warn	NT	St.out	Rec	Ret	C.arg	C.inv
Polyfill.io (map)	0	0	0	0	0	0	0	0	0	0	0	0	6	27	7	0
Polyfill.io (find)	0	0	0	0	0	0	0	0	0	0	0	0	0	9	19	19
Mozilla (filter)	0	0	0	0	5	7	5	0	0	0	0	0	17	35	16	0
es5-shim	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bluebird	7	0	0	1	1	1	262	252	211	0	0	0	0	0	316	276
Q	57	870	0	0	0	0	3	0	149	504	68	0	0	0	2	0
when	5	2	0	1	1	1	45	17	207	123	0	0	0	0	165	112
then/promise	57	870	0	0	0	0	0	0	149	504	0	0	0	0	1	0
rsvp.js	57	870	0	0	0	0	23	0	149	504	0	0	0	0	21	0
native-promise-only	57	870	0	0	0	0	47	0	149	504	0	0	0	0	48	0
lie	57	870	0	0	0	0	27	0	149	504	0	0	0	0	22	0
pacta	57	870	0	0	0	0	0	0	149	504	0	0	0	0	0	0
es6-promises	57	870	0	0	0	0	0	0	149	504	0	0	0	0	0	0
bloodhound-promises	57	870	0	0	0	0	106	1	149	504	0	0	0	0	51	0

Table 6. Behavioral differences in array polyfills generated by Mimic [Heule et al. 2015].

Polyfill	Base	Cb-Empty	Cb-Quick	Cb-Mined	Cb-Writes
Every	1,000	1,000	999	437	992
Some	1,000	1,000	993	195	974
ForEach	1,000	0	0	0	0
Filter	1,000	0	0	0	7
Map	1,000	0	0	0	28
Reduce	1,000	797	794	758	526
ReduceRight	1,000	797	794	783	809

Future work might combine a test generator for higher-order functions, such as *LambdaTester*, with an approach for synthesizing polyfills, such as Mimic, so that the behavioral differences found with generated tests provide feedback on weaknesses of the synthesized code.

6.5 Examples of Bugs and Other Inconsistencies

To illustrate the behavioral differences detected by *LambdaTester*, we discuss a few representative examples. For space reasons, we only include the relevant fragments of the generated tests that expose errors.

Unexpected types and number of arguments. Figure 6a illustrates a test case that exposes a behavioral difference found in the *when* library caused by passing a non-function value as the second argument to the `catch` method. The buggy polyfill implementation is given in Figure 6b.

The library throws a `TypeError` because it tries to execute the `call` method on a non-function object at line 83. In contrast, the native implementation ignores the second argument and executes

```
var p1 = Promise.resolve(18);
var p2 = Promise.reject(17);
p2.catch(function(){
  return p1;
}, p1);
```

(a) *LambdaTester*-generated test.

```
Promise.prototype['catch'] = function(onRejected) {
  if (arguments.length < 2) {
    return origCatch.call(this, onRejected);
  }
  if (typeof onRejected !== 'function') {
    return this.ensure(rejectInvalidPredicate);
  }
  return origCatch.call(this,
  createCatchFilter(arguments[1],onRejected));
}

function createCatchFilter(handler, predicate) {
  return function(e) {
    return evaluatePredicate(e, predicate) ?
        handler.call(this, e) : reject(e);
  }
}
```

(b) Implementation of `catch` from *when* library.

Fig. 6. Example of behavioral difference found in the *when* library.

```
86 var p1 = Promise.resolve(18);
87 var p2 = Promise.reject(17);
88 var r1 = p1.then(function(){ return null; }, null);
89 var r2 = p2.then(function(){ return r1; });
90 var r3 = r2.catch(function(){ return p2; });
91 var r4 = r1.then(function(){ return p2; });
```

Fig. 7. Example of *LambdaTester*-generated test that exposes a behavioral difference in the *Q* library.

the method call without errors. This example illustrates a situation where a different output is written to the standard error stream.

Order of executed function calls. The example in Figure 7 illustrates a behavioral difference found in the *Q* library. The native implementation first executes calls to the `then` function at lines 88, 89, and 91 and then a call to `catch` at line 90. However, the *Q* library executes the method calls in the same order as presented in Figure 7. *LambdaTester* discovers this difference by inspecting the state of the callback arguments.

The cause of this bug appears to be in the library's queuing mechanism used for tracking unhandled rejections. Due to the complexity of the code, we were not able to fully diagnose the problem. In general, the complexity of callback-based code is a good reason for extensive testing, e.g., using our *LambdaTester* approach.

Changing receiver object inside a callback. Many of the behavioral differences found in the array polyfills are caused by changes made by the callback to properties of the receiver object or of other arguments. Figure 8a illustrates a test case that expose this type of problem, with the corresponding polyfill implementation in Figure 8b. In the test, when the method under test invokes the callback for the first time, the callback sets the `length` property of the receiver object to `false`. At line 106 in Figure 8b, the method under test performs a check to find whether the receiver object has a

```
var base = ["w", "I", 126];
base.find(function(a,b,c){
  base['length'] = false;
  return a;
});
```

(a) *LambdaTester*-generated test

```
function find(callback) {
  ...
  var object = Object(this),
  scope = arguments[1],
  arraylike = object instanceof String ?
      object.split('') : object,
  index = -1;

  while (++index < length) {
    if (index in arraylike) {
      element = arraylike[index];
      if (callback.call(scope, element, index, object) {
        return element;
      }
    }
  }
}
```

(b) Implementation of `Array.prototype.find` from polyfill.io.

Fig. 8. Example of a behavioral difference found in the *polyfill.io* library.

property named `index`. After changing the `length` property, the check always evaluates to `false`, which prevents further executions of the callback argument. As a result, the polyfill and the native implementation return the same value, `undefined`, but the native implementation executes the callback three times, whereas the polyfill executes it only once. This example illustrates a situation where a callback-related oracle helps detect behavioral differences that would be missed otherwise.

Despite the effectiveness of *LambdaTester* in detecting inconsistencies, we are aware that the developers of the tested libraries may find some of the generated tests more useful than others. The reason is that not every generated test represents a realistic usage scenario of the tested libraries. For example, method invocations with callbacks that change the state of the receiver or of the argument objects are unlikely to be a common use case. However, testing uncommon behavior helps finding more bugs that would be missed otherwise.

6.6 Effectiveness in Covering Code Under Test

To assess the effectiveness of *LambdaTester* in covering code under test we measure statement coverage. For the array polyfills, we collect coverage data for each method implementation. However, since it is not straightforward to extract individual method implementations from the promise libraries, we chose to measure coverage of the entire promise libraries.

Table 7 lists the results for coverage measurements for each benchmark. The statement coverage of polyfills differs significantly between *Base* and the feedback-directed approaches, e.g., increasing from 20% to up to 100% for the `filter` method from the Mozilla library. Overall, *Cb-Writes* is the most effective approach in increasing statement coverage. For all benchmarks, feedback-directed approaches achieve better statement coverage compared to the baseline approach.

The statement coverage for promise libraries is relatively low, and the reason is that the tested methods comprise only a subset of the entire library code: In addition to the `then` and `catch` methods, the promise libraries define other functions not targeted by our generated tests.

Table 7. Statement coverage for 1,000 generated tests.

Benchmark	Base	Cb-Empty	Cb-Quick	Cb-Mined	Cb-Writes
Polyfill.io (every)	40.0%	70.0%	80.0%	80.0%	90.0%
Polyfill.io (some)	40.0%	70.0%	80.0%	80.0%	90.0%
Polyfill.io (forEach)	44.4%	77.7%	77.7%	77.7%	88.8%
Polyfill.io (filter)	36.3%	72.7%	81.8%	81.8%	90.9%
Polyfill.io (map)	40.0%	80.0%	80.0%	80.0%	90.0%
Polyfill.io (find)	36.3%	72.7%	81.8%	81.8%	90.9%
Polyfill.io (reduce)	25.0%	81.2%	87.5%	81.2%	87.5%
Polyfill.io (reduceRight)	25.0%	81.2%	87.5%	81.2%	87.5%
Mozilla (every)	35.0%	80.0%	90.0%	90.0%	95.0%
Mozilla (some)	33.3%	75.0%	83.3%	83.3%	91.6%
Mozilla (forEach)	41.1%	88.2%	88.2%	88.2%	94.1%
Mozilla(filter)	20.0%	80.0%	93.3%	93.3%	100%
Mozilla (map)	35.0%	90.0%	90.0%	90.0%	95.0%
Mozilla (find)	40.0%	80.0%	86.6%	86.6%	93.3%
Mozilla (reduce)	19.0%	80.9%	85.7%	80.9%	85.7%
Mozilla (reduceRight)	23.5%	76.4%	82.3%	76.4%	82.3%
es5-shim (every)	56.5%	58.9%	59.6%	59.6%	63.5%
es5-shim (some)	56.5%	58.9%	59.6%	59.6%	63.5%
es5-shim (forEach)	56.4%	60.3%	60.3%	60.3%	64.1%
es5-shim (filter)	56.3%	60.1%	60.9%	60.9%	64.6%
es5-shim (map)	56.0%	60.6%	60.6%	60.6%	64.3%
es5-shim (reduce)	48.8%	58.6%	59.4%	58.6%	62.4%
es5-shim (reduceRight)	48.1%	59.2%	60.0%	59.2%	62.9%
Q	42.2%	43.8%	43.8%	44.2%	43.8%
bluebird	37.2%	39.6%	39.9%	40.0%	41.0%
when	51.5%	52.5%	52.8%	52.8%	53.2%
then/promise	48.9%	59.0%	62.1%	63.6%	64.6%
rsvp.js	41.9%	45.3%	46.5%	47.2%	47.9%
native-promise-only	65.1%	67.4%	68.0%	68.6%	69.1%
lie	41.2%	55.5%	55.5%	57.1%	62.3%
pacta	39.3%	54.3%	55.9%	56.6%	59.0%
es6-promises	58.6%	67.2%	68.8%	68.8%	68.8%
bloodhound-promises	29.9%	33.0%	33.3%	35.3%	36.4%

6.7 Efficiency

To assess the performance of the test generation techniques, Table 8 shows, for each approach, the time needed to generate and execute 1,000 tests. All experiments are conducted on a 48-core machine with a 2.2GHz Intel Xeon CPU and 64GB of RAM. We use Node.js 8.5 and provide it with the default of 1GB of memory. The implementation of *LambdaTester* is single-threaded and while running the tool we effectively use a single core.

The execution time of the *Base* approach is dominated by the time needed to generate tests and collect their execution summaries. The execution time of the feedback-directed approaches is higher as they also include the time to generate multiple calls and to collect feedback. In particular, for the *Cb-Writes* approach, the time needed for dynamically analyzing each test's execution dominates the total execution time. The time spent to generate tests with *Cb-Writes* takes less than 1 hour on average, except for the generation of the promise tests, which takes approximately 3 hours. Overall, since running *LambdaTester* requires minimal manual intervention and since the generated tests expose many behavioral differences, we consider the computational effort to be acceptable.

Table 8. Time to generate 1,000 tests per API.

API	Base	Cb-Empty	Cb-Quick	Cb-Mined	Cb-Writes
every	6m 50s	27m 6s	27m 5s	33m 35s	53m 35s
forEach	6m 48s	27m 6s	27m 5s	33m 37s	53m 46s
some	6m 48s	27m 6s	27m 6s	33m 37s	53m 26s
filter	6m 47s	27m 6s	27m 6s	33m 34s	53m 7s
map	6m 47s	27m 6s	27m 6s	32m 30s	53m 34s
reduce	6m 46s	27m 5s	27m 7s	33m 1s	54m
reduceRight	6m 46s	27m 5s	27m 7s	33m 39s	53m 54s
find	6m 46s	27m 6s	27m 7s	33m 33s	53m 34s
then,catch	6m 31s	27m 10s	27m 6s	33m 23s	190m

6.8 Summary of Results

We summarize our findings as follows:

RQ1: Effectiveness in finding behavioral differences. LambdaTester finds behavior differences in 12 out of 13 libraries. When comparing the different techniques for creating callbacks, we find that our novel *Cb-Writes* approach is the most effective.

RQ2: Kinds of behavioral differences. LambdaTester detects a diverse set of differences, including clearly undesired behavior, such as non-termination bugs and crashes in polyfills, as well as differences in the number of times that a callback gets invoked.

RQ3: Effectiveness in increasing code coverage. The *Cb-Writes* is the most effective approach for increasing the statement coverage of the array and promise polyfills.

RQ4: Efficiency. The time required by *LambdaTester* to generate a single test ranges between 0.4 and 12 seconds, making it a practical tool for automatically testing higher-order functions.

7 RELATED WORK

7.1 Test Generation

Random testing has been shown to be surprisingly cost-effective [Duran and Ntafos 1984; Ntafos 2001] and to detect a predictable number of bugs despite its random nature [Ciupa et al. 2011], providing an effective prelude to more rigorous bug detection techniques [Groce et al. 2007]. Adaptive random testing [Chen et al. 2010, 2004] tries to increase the effectiveness of testing by equally distributing test inputs across the input domain, but appears to be less cost-effective than random testing [Arcuri and Briand 2011; Ciupa et al. 2008]. Random test generators include JCrasher [Csallner and Smaragdakis 2004], which creates random arguments guided by test annotations, and Randoop [Pacheco and Ernst 2007; Pacheco et al. 2008], which uses feedback from executions of previously generated partial tests. *LambdaTester*, like Randoop, performs a form of feedback-directed, random test generation, and adds the ability to effectively test higher-order functions. A test generator by Zheng et al. considers the object fields modified by the code under test to select which methods to call [Zheng et al. 2010]. Similarly, our *Cb-Writes* technique analyzes reads in the code under test, to direct the generation of callback functions toward writing to those locations.

Some test generators for functional languages address the problem of testing higher-order functions. QuickCheck [Claessen and Hughes 2011] randomly generates functions that return a type-correct return value. However, the generated functions do not modify any other state beyond the return value. Koopman and Plasmeijer propose to improve QuickCheck by systematically generating functions based on the AST representation of a function argument [Koopman and Plasmeijer 2006]. The basic idea of their approach is to represent functions as a data type and to systematically enumerate elements of this data type. In contrast to the *Cb-Writes* variant of *LambdaTester*, their approach does not generate callback bodies specifically targeted at the code under test. Instead, the user of the approach needs to provide a generator function for callback bodies, which, e.g., creates expressions to be used in the body. For an empirical comparison with a QuickCheck implementation for JavaScript, see Section 6.

Two test generators for Racket tests higher-order functions, either by generating new subclasses of existing classes, guided by the types of functions and the environment, and guided by developer-provided contracts [Klein et al. 2010], or by adopting symbolic execution for higher-order functions [Nguyen and Horn 2015]. Another line of work generates functions to test a Haskell compiler [Palka et al. 2011]. Their focus is on generating type-correct functions, whereas we focus on generating functions that trigger diverse behaviors. All the above test generators are guided by static type signatures, which are not available in the dynamic languages that we target here. Another difference is that, in contrast to our *Cb-Writes* technique, none of the above approaches guides the generation of functions based on feedback from the code under test.

Symbolic testing [Cadar et al. 2008; King 1976; Visser et al. 2004; Xie et al. 2005] and concolic testing [Godefroid et al. 2005, 2008; Sen et al. 2005] are another popular form of test generation. Other test generation techniques include bounded exhaustive test generation guided by pre- and post-conditions [Boyapati et al. 2002], test generation combined with mining of call sequences from existing code [Thummalapenta et al. 2009], combining symbolic testing with static analysis [Thummalapenta et al. 2011], UI-level test generation [Artzi et al. 2011; Ermuth and Pradel 2016; Marchetto et al. 2008; Memon 2007; Mesbah et al. 2008], and performance-guided test generation [Burnim et al. 2009; Pradel et al. 2014]. To the best of our knowledge, none of these approaches addresses the problem of testing higher-order functions.

7.2 Testing and Analysis for JavaScript

The dynamic nature of JavaScript makes code prone to errors and other undesired code properties. Several dynamic analyses [Sen et al. 2013] have been proposed to detect such problems, including analyses to detect type inconsistencies [Pradel et al. 2015], code quality problems [Gong et al. 2015b], JIT compilation-related performance bottlenecks [Gong et al. 2015a], memory leaks [Jensen et al. 2015], data races [Mutlu et al. 2015], and conflicts between libraries that share the global namespace [Patra et al. 2018]. Other dynamic analyses help developers understand the dynamic behavior of callback-heavy JavaScript code [Alimadadi et al. 2014] and to prevent unintended information flows [Chugh et al. 2009; Hedin et al. 2014]. A recent survey [Andreasen et al. 2017] provides a comprehensive summary of dynamic analyses for JavaScript. All dynamic analyses rely on inputs to drive the execution, a problem that test generators such as *LambdaTester* address.

A recent test generator for JavaScript creates tests that check TypeScript interface declarations against the corresponding JavaScript implementations [Kristensen and Møller 2017]. Their approach provides only rudimentary support for higher-order functions by passing "a simple dummy function" [Kristensen and Møller 2017], similar to our Cb-Empty technique.

Beyond test generation and dynamic analysis, previous research has focused on type systems [Jensen et al. 2009; Thiemann 2005] and several static analyses for JavaScript, including points-to analysis [Sridharan et al. 2012], a memory leak detector [Pienaar and Hundt 2013], an analysis of library clients [Madsen et al. 2013], an analysis of event-driven Node.js code [Madsen et al. 2015], and checks of JavaScript libraries against their corresponding TypeScript interface declarations [Feldthaus and Møller 2014]. Given the difficulties of analyzing an inherently dynamic language such as JavaScript, dynamic analysis is a viable alternative for which *LambdaTester* can provide inputs.

8 CONCLUSIONS

We present a framework for testing higher-order functions in dynamic programming languages. The approach consists of two phases: the *discovery phase* is concerned with discovering at which argument positions a function is expected, and the *test generation phase* automatically creates tests that perform a sequence of method calls. The created method calls pass callbacks at those positions where they are expected and random values at others. Both phases take as input setup code that creates a set of initial values that are used as receivers and arguments in subsequently generated calls.

We have implemented the framework in a tool called *LambdaTester*, and evaluate several instances of the framework in which the generated callback functions consist of: (i) empty functions, (ii) functions that return random values [Claessen and Hughes 2011], (iii) callbacks mined from a corpus of existing code, and (iv) functions that write to locations that are likely to be read, as determined using a feedback-directed dynamic analysis technique. We apply *LambdaTester* to polyfills for array-related functions taken from the *es5-shim*, *mozilla*, and *polyfill.io* libraries, and to polyfills of the promise APIs taken from *bluebird*, *Q*, and *when*. Our experimental results show that *LambdaTester* reveals various behavioral differences between polyfills and their corresponding native implementations, including previously unknown bugs in popular polyfills. Overall, the approach detects differences in 12 of 13 libraries. Comparing the different techniques for creating callback functions shows that generating callbacks that modify program state in non-obvious ways is more likely to expose behavioral diferences than the simpler approaches, and that our novel feedback-directed technique exposes behavioral differences missed by all other techniques.

ACKNOWLEDGMENTS

This work was supported by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, by the German Research Foundation within the ConcSys and Perf4JS projects, and by the Hessian LOEWE initiative within the Software-Factory 4.0 project. This research was also supported in part by NSF grant CCF-1715153.

REFERENCES

Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. 2014. Understanding JavaScript event-based interactions. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014.* 367–377.

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).

Andrea Arcuri and Lionel Briand. 2011. Adaptive Random Testing: An Illusion of Effectiveness?. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11).* ACM, New York, NY, USA, 265–275. https://doi.org/10.1145/2001420.2001452

Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. 2011. A framework for automated testing of JavaScript web applications. In *ICSE.* 571–580.

Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis (ISSTA).* 123–133.

Jacob Burnim, Sudeep Juvekar, and Koushik Sen. 2009. WISE: Automated test generation for worst-case complexity. In *ICSE.* IEEE, 463–473.

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Symposium on Operating Systems Design and Implementation (OSDI).* USENIX, 209–224.

Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. 2010. Adaptive Random Testing: The ART of test case diversity. *Journal of Systems and Software* 83, 1 (2010), 60–66. https://doi.org/10.1016/j.jss.2009.02.022

Tsong Yueh Chen, Hing Leung, and IK Mak. 2004. Adaptive random testing. In *Annual Asian Computing Science Conference.* Springer, 320–329.

Ravai Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. 2009. Staged Information Flow for JavaScript. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 50–62.

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.

I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, and B. Meyer. 2011. On the number and nature of faults found by random testing. *Software Testing, Verification and Reliability* 21, 1 (2011), 3–28. https://doi.org/10.1002/stvr.415 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.415

Koen Claessen and John Hughes. 2011. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 46, 4 (May 2011), 53–64. https://doi.org/10.1145/1988042.1988046

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

Joe W. Duran and Simeon C. Ntafos. 1984. An Evaluation of Random Testing. *IEEE Trans. Softw. Eng.* 10, 4 (July 1984), 438–444. https://doi.org/10.1109/TSE.1984.5010257

Markus Ermuth and Michael Pradel. 2016. Monkey See, Monkey Do: Effective Generation of GUI Tests with Inferred Macro Events. In *International Symposium on Software Testing and Analysis (ISSTA)*. 82–93.

Asger Feldthaus and Anders Møller. 2014. Checking correctness of TypeScript interfaces for JavaScript libraries. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, 1–16.

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.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. *SIGPLAN Not.* 40, 6 (June 2005), 213–223. https://doi.org/10.1145/1064978.1065036

Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security Symposium (NDSS)*.

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.

Alex Groce, Gerard Holzmann, and Rajeev Joshi. 2007. Randomized Differential Testing As a Prelude to Formal Verification. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 621–631. https://doi.org/10.1109/ICSE.2007.68

Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: tracking information flow in JavaScript and its APIs. In *SAC*. 1663–1671.

Stefan Heule, Manu Sridharan, and Satish Chandra. 2015. Mimic: computing models for opaque code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 710–720.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Symposium on Static Analysis (SAS)*. Springer, 238–255.

Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. 2015. MemInsight: platform-independent memory debugging for JavaScript. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 345–356.

J. C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

Casey Klein, Matthew Flatt, and Robert Bruce Findler. 2010. Random testing for higher-order, stateful programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 555–566.

Pieter Koopman and Rinus Plasmeijer. 2006. Automatic Testing of Higher Order Functions. In *Asian Symposium on Programming Languages and Systems (APLAS)*. 148–164.

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

Magnus Madsen, Benjamin Livshits, and Michael Fanning. 2013. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *ESEC/SIGSOFT FSE*. 499–509.

Magnus Madsen, Frank Tip, and Ondrej Lhoták. 2015. Static analysis of event-driven Node.js JavaScript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 505–519.

Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. 2008. State-Based Testing of Ajax Web Applications. In *ICST*. IEEE Computer Society, 121–130.

William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

Atif M. Memon. 2007. An event-flow model of GUI-based applications for testing. *Softw. Test., Verif. Reliab.* (2007), 137–157.

Ali Mesbah, Engin Bozdag, and Arie van Deursen. 2008. Crawling Ajax by Inferring User Interface State Changes. In *International Conference on Web Engineering (ICWE)*. 122–134.

Ali Mesbah and Arie van Deursen. 2009. Invariant-based automatic testing of Ajax user interfaces. In *ICSE*. 210–220.

Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. 2015. Detecting JavaScript Races that Matter. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*.

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.

S. C. Ntafos. 2001. On comparisons of random, partition, and proportional partition testing. *IEEE Transactions on Software Engineering* 27, 10 (Oct 2001), 949–960. https://doi.org/10.1109/32.962563

Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion (OOPSLA '07)*. ACM, New York, NY, USA, 815–816. https://doi.org/10.1145/1297846.1297902

Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. 2008. Finding Errors in .Net with Feedback-directed Random Testing. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, New York, NY, USA, 87–96. https://doi.org/10.1145/1390630.1390643

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 '11)*. ACM, New York, NY, USA, 91–97. https://doi.org/10.1145/1982595.1982615

Jibesh Patra, Pooja N. Dixit, and Michael Pradel. 2018. ConflictJS: Finding and Understanding Conflicts Between JavaScript Libraries. In *ICSE*.

Jacques A. Pienaar and Robert Hundt. 2013. JSWhiz: Static analysis for JavaScript memory leaks. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*. 11:1–11:11.

Michael Pradel and Thomas R. Gross. 2012. Fully Automatic and Precise Detection of Thread Safety Violations. In *Conference on Programming Language Design and Implementation (PLDI)*. 521–530.

Michael Pradel, Parker Schuh, George Necula, and Koushik Sen. 2014. EventBreak: Analyzing the Responsiveness of User Interfaces through Performance-Guided Test Generation. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 33–47.

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

Veselin Raychev, Pavol Bielik, Martin T. Vechev, and Andreas Krause. 2016. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 761–774.

Malavika Samak and Murali Krishna Ramanathan. 2014. Multithreaded Test Synthesis for Deadlock Detection. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 473–489.

Marija Selakovic, Thomas Glaser, and Michael Pradel. 2017. An Actionable Performance Profiler for Optimizing the Order of Evaluations. In *International Symposium on Software Testing and Analysis (ISSTA)*. 170–180.

Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 488–498. https://doi.org/10.1145/2491411.2491447

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation Tracking for Points-To Analysis of JavaScript. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*. 435–458.

Peter Thiemann. 2005. Towards a Type System for Analyzing JavaScript Programs. In *European Symposium on Programming (ESOP)*. 408–422.

Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. MSeqGen: Object-oriented unit-test generation via mining source code. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 193–202.

Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. 2011. Synthesizing method sequences for high-coverage testing.. In *OOPSLA*. 189–206.

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.

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.

Wujie Zheng, Qirun Zhang, Michael Lyu, and Tao Xie. 2010. Random Unit-test Generation with MUT-aware Sequence Recommendation. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*. ACM, New York, NY, USA, 293–296. https://doi.org/10.1145/1858996.1859054