162

Finding Broken Promises in Asynchronous JavaScript Programs

SABA ALIMADADI, Northeastern University, USA
DI ZHONG, Northeastern University, USA
MAGNUS MADSEN, Aarhus University, Denmark
FRANK TIP, Northeastern University, USA

Recently, promises were added to ECMAScript 6, the JavaScript standard, in order to provide better support for the asynchrony that arises in user interfaces, network communication, and non-blocking I/O. Using promises, programmers can avoid common pitfalls of event-driven programming such as event races and the deeply nested counterintuitive control flow referred to as “callback hell”. Unfortunately, promises have complex semantics and the intricate control– and data-flow present in promise-based code hinders program comprehension and can easily lead to bugs. The *promise graph* was proposed as a graphical aid for understanding and debugging promise-based code. However, it did not cover all promise-related features in ECMAScript 6, and did not present or evaluate any technique for constructing the promise graphs.

In this paper, we extend the notion of promise graphs to include all promise-related features in ECMAScript 6, including default reactions, exceptions, and the synchronization operations `race` and `all`. Furthermore, we report on the construction and evaluation of PromiseKeeper, which performs a dynamic analysis to create promise graphs and infer common promise anti-patterns. We evaluate PromiseKeeper by applying it to 12 open source promise-based Node.js applications. Our results suggest that the promise graphs constructed by PromiseKeeper can provide developers with valuable information about occurrences of common anti-patterns in their promise-based code, and that promise graphs can be constructed with acceptable run-time overhead.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Concurrent programming structures*;

Additional Key Words and Phrases: JavaScript, Promises, Promise Graph, Dynamic Analysis, PromiseKeeper

ACM Reference Format:
Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. 2018. Finding Broken Promises in Asynchronous JavaScript Programs. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 162 (November 2018), 26 pages. https://doi.org/10.1145/3276532

1 INTRODUCTION

Recently, promises where added to ECMAScript 6 [Ecmascript 2015], the JavaScript standard, in order to provide better support for the asynchrony that arises in user-interfaces, network communication, and non-blocking I/O. Such asynchrony has traditionally been implemented using event-driven programming, which leads to awkward nonlinear control flow commonly referred to as “callback hell”. Event-driven programming also provides poor support for error handling, and unless programmers are careful, event-driven programs may experience non-deterministic failures

Authors' addresses: Saba Alimadadi, Northeastern University, Boston, MA, USA, saba@northeastern.edu; Di Zhong, Northeastern University, Boston, MA, USA, zhong.d@husky.neu.edu; Magnus Madsen, Aarhus University, Aarhus, Denmark, magnusm@cs.au.dk; Frank Tip, Northeastern University, Boston, MA, USA, f.tip@northeastern.edu.

2475-1421/2018/11-ART162
https://doi.org/10.1145/3276532

due to event race errors [Adamsen et al. 2017b,a; Petrov et al. 2012; Raychev et al. 2013; Zhang and Wang 2017; Zheng et al. 2011], and lost events and dead listeners [Madsen et al. 2015] that are hard to debug.

Promises aim to overcome these problems by providing an abstraction for the result of an asynchronous computation. A promise is in one of three states: pending, fulfilled, or rejected. A pending promise has not yet been settled, i.e., resolved or rejected with a value. A fulfilled promise holds the result of an asynchronous computation that has successfully completed, whereas a rejected promise holds an error value of an asynchronous computation that somehow failed. Once a promise has been fulfilled or rejected, its value cannot change, i.e. a promise can only be settled once. Each promise object is equipped with two functions, `resolve` and `reject`, that are used to to fulfill or reject the promise. Promises enable programmers to compose asynchronous computations by associating *reactions* with a promise. When a promise is resolved or rejected, the corresponding reaction is executed and the resulting value is wrapped in another promise, enabling programmers to create a *chain* of asynchronous computations where each computation depends on the value of the previous computation. Promises support proper error-handling by allowing errors to be propagated along these promise chains.

Unfortunately, promises are complex in their own right and JavaScript programmers are easily confused by their semantics, causing them to make mistakes that result in hard-to-debug errors [Madsen et al. 2015]. For example, programmers may forget to resolve or reject a promise on all paths through the program, or forget to register a resolve and reject reaction on a promise. Other common mistakes are situations where a promise is unintentionally resolved with the value `undefined` when a function that was registered as a reaction returns implicitly, where an attempt is made to resolve or reject a promise that was already settled, or where programmers unintentionally construct a promise chain that has an fork.

Prior work [Madsen et al. 2017] provided a formal semantics for a core subset of JavaScript promises, and proposed the *promise graph* as a visual aid for understanding and debugging promise-based programs. This work argued for the usability of promise graphs, but it did not handle all promise-related features in ECMAScript 6, and it did not present any technique for computing promise graphs. Moreover, its evaluation was limited to a case study in which promise graphs were constructed manually for small program fragments taken from the StackOverflow website.

In this paper, we present an extension of the promise graph to handle all promise-related features of the ECMAScript 6 standard, including exceptions, default reactions, and the `race` and `all` constructs that are used for synchronization on multiple promises. We report on the implementation of a tool, PromiseKeeper, that automatically constructs promise graphs based on dynamic analysis. In an empirical evaluation, we apply PromiseKeeper to 12 promise-based Node.js applications taken from GitHub. Our findings show that PromiseKeeper is capable of constructing promise graphs for large and complex applications with acceptable run-time overhead. Furthermore, we show that PromiseKeeper is able to detect anti-patterns such as missing reject reactions, attempts to settle a promise multiple times, unsettled promises, unnecessary promises, implicit returns in reactions, and unreachable reactions that warrant further investigation by a developer. Such anti-patterns manifest questionable coding practices that are often, but not always correlated with bugs [Gamma 1995]. We convey these findings using a visual representation that enables developers to quickly obtain an understanding of the behavior of, and identify potential problems in promise-based code.

In summary, this paper makes the following contributions:

- We extend the promise graph proposed by Madsen et al. [2017] to handle all aspects ECMAScript 6 promises, including default reactions, exceptions, and the synchronization operations `all` and `race`.
- We implement a dynamic analysis in an open-source tool called PromiseKeeper to automatically construct the promise graph for a specific execution [PromiseKeeper 2018]. We also discuss how PromiseKeeper can be used to detect common anti-patterns related to promises.
- We show that PromiseKeeper is capable of producing promise graphs for realistic applications, and identify instances of anti-patterns that are known to cause problems in promise-based code.

The remainder of the paper is structured as follows. Section 2 provides a brief primer on JavaScript promises. Next, in Section 3 we use three examples taken from GitHub to illustrate the problems that can arise in promise-based code. Section 4 defines the promise graph, as an extension of the promise graphs of Madsen et al. [2017]. Section 5 discusses how the promise graph can be used to aid debugging by detecting occurrences of several anti-patterns. Section 6 describes the dynamic analysis implemented in PromiseKeeper. Section 7 presents the evaluation of PromiseKeeper. Finally, related work is presented in Section 8 and Section 9 concludes.

2 REVIEW OF JAVASCRIPT PROMISES

This section presents a brief review of JavaScript promises. Readers already familiar with this feature may skip to the next section.

A *promise* represents the result of an asynchronous computation, and is in one of three states: "pending", "fulfilled", or "rejected". Upon creation, a promise is in the pending state, from which it can transition to the fulfilled state by invoking a function `resolve` with a result value, or it can transition to the rejected state by invoking a function `reject` with an error value. A promise that is in the fulfilled or rejected state is also referred to as being "settled". Once settled, the state of a promise cannot change again.

Creating promises. Creating a promise is accomplished by invoking the `Promise` constructor with callback that takes two functions, `resolve` and `reject`, as arguments. The body of this function may invoke these functions to resolve or reject the promise with a specific value, respectively. For example, the following code:

```
var p0 = new Promise(function(resolve, reject) {
  if (Math.random() > 0.5) {
    resolve(17);
  } else {
    reject(new Error('An error has occurred'));
  }
});
```

randomly decides to fulfill the promise with the value 17, or to reject it with an `Error` object.

Alternatively, promises may be created and immediately resolved with a value by calling the function `Promise.resolve`, or created and immediately rejected with a value by calling the function `Promise.reject`. For example, the expression `Promise.resolve(17)` creates a promise that is immediately resolved with the value 17.

Registering reactions. Programmers can register *reactions* on promises using the method `then`. Continuing the above example, the following code:

```
p0.then(function f1(v){
  console.log('value: ' + v);
```

```
    return v + 1;
  }, function f2(e){
    console.log('error: ' + e);
    throw e;
  });
  console.log('done');
```

defines the function `f1` to be a *fulfill reaction* for the promise in variable `p0`, and the function `f2` to be a *reject reaction* for `p0`. This means that, if `p0` is resolved, `f1` will be invoked with the value that `p0` was resolved with, and that, if `p0` was rejected, then `f2` will be invoked with the value that `p0` was rejected with. Note that only one of these reactions will execute, because a promise can be settled only once. Furthermore, it is important to realize that the execution of a reaction will not take place until control flow returns to the main event loop. In other words, the above code first prints "done", followed by "`value: 17`" or "`error: foo`" depending on whether `p0` was resolved or rejected. The above example showed a situation where `then` was invoked with two arguments representing the resolve reaction and reject reaction. However, it is possible to omit the reject reaction as we shall see in the next example.

Creating a chain of promises. Each call to the `then` method creates a new promise. If the reaction that executed returned a value, the returned value is used to resolve that promise. If the reaction that executed threw an exception, the thrown value is used to reject that promise. This enables the creation of a *chain* of asynchronous computations. For example, the following code fragment:

```
var p0 = Promise.resolve(17);
p0.then(function g1(v){ return v + 1; })
  .then(function g2(v){ return v + 1; })
  .then(function g3(v){ console.log(v); })
```

creates 4 promises that are resolved with the values `17`, `18`, `19`, and `undefined`, respectively. Note that, since the function `g3` does not explicitly return a value, it implicitly returns the value `undefined`, causing the last promise in the chain to be resolved with the value `undefined`.

Error handling. So far, we have seen how `then` can be used to register fulfill and reject reactions with a promise. A method `catch` can be used to register a reject reaction on a promise. For example, in the following code fragment:

```
var p0 = Promise.reject('foo');
p0.catch(function f(e){
  console.log('error: ' + e);
});
```

a promise `p0` is rejected with the value `foo`, causing the function `f` to be executed, which will print the message "`error: foo`".

One of the key advantages of promises over traditional event-based programming is the fact that errors propagate through a chain of asynchronous computations. This is accomplished using a mechanism that we will refer to as *default reactions*: An expression `p.then(f)` that only specifies a fulfill reaction has an implicitly defined default reject reaction `function(v){ throw v; }`.[1] This means that programmers can write:

```
var p0 = new Promise(function(resolve, reject){ ... };
p0.then(function(v){ ... })
  .then(function(v){ ... })
  .then(function(v){ ... })
```

[1] Likewise, an expression `p.catch(f)` has an implicit default fulfill reaction `function(v){ return v; }`.

```
  .catch(function(err){ ... })
```

where a single `catch` at the end of a promise chain suffices to handle errors that occur during any of the preceding computations.

It is possible to pass non-function arguments to `then` and `catch` as reactions. These are interpreted as default reactions. For example, the following code:

```
var p0 = Promise.resolve(17);
p0.then(undefined)
  .then(function(v){ console.log(v); });
```

prints "17" because the value `undefined` is interpreted as the default reaction `function(v){ return v; }`.

Linking. When an application that performs asynchronous computations needs to interact with other components that use promises, it is helpful to link the state of two promises. This can be accomplished by having a function `f` that is used as a reaction return a promise p (as opposed to some other kind of value)[2]. In such cases, the state of the promise p' created by an expression `p.then(f)` becomes *linked* with the state of p. In particular, if/when p is resolved with value v, then p' will be resolved with value v, and if p is rejected with value e, then p' will be rejected with value e as well. In other words, in the following code fragment:

```
var p0 = Promise.resolve(17);
var p1 = Promise.reject("foo");
p0.then(function(v){ return p1; });
```

the promise created on the last line is rejected with the value `foo`.

`all` *and* `race`. The JavaScript promises API supports two synchronization operations, `all` and `race`, that enable programmers to perform synchronization actions involving multiple promises. In particular, given an array a of promises `[`$p_0, \cdots, p_n$`]`, `Promise.all(`a`)` returns a promise that is fulfilled if all of the promises p_i are fulfilled, and rejected if at least one of the promises p_i is rejected. Furthermore, `Promise.race(`a`)` returns a promise that is fulfilled or rejected as soon as one of the promises p_i is fulfilled or rejected.

3 CHALLENGES AND MOTIVATION

JavaScript developers utilize Promises to remedy the error handling and nesting problems associated with asynchronous callbacks. However, the complex semantics of promise-based code can impede comprehension of its dynamic behavior. As a result, developers often make mistakes while creating, settling, linking, and synchronizing various promise objects. As such, common error-prone patterns manifest in promise-based JavaScript applications [Madsen et al. 2017]. We use three examples, presented in Figures 1–6, to illustrate the challenging nature of understanding and debugging promise-based code.

3.1 Unhandled Promise Rejections

A promise is typically fulfilled or rejected by an invocation of the `resolve` and `reject` functions associated with a promise. However, a promise might also be *imlicitly* rejected when an exception is thrown by a promise reaction. Such uncaught exceptions are not propagated through regular exception handling mechanism of JavaScript, but instead cause the dependent promise to be rejected.

[2] Similarly, promises can be linked at promise creation time by resolving a promise with another promise. This feature was omitted from the semantics by Madsen et al. [Madsen et al. 2017].

```
35 tcpPortUsed.check(port, 'localhost').then(function(inUse) {
36     if (inUse) {
37         console.error(port + ' is in use');
38         process.exit(1);
39     } else {
40         console.log('Starting Docusaurus server on port ' + port + '...');
41         // start local server on specified port
42         var server = require('./server/server.js');
43         server(port);
44     }
45 });
```

Fig. 1. JavaScript code fragment from `start-server.js` in Docusaurus.

Thus, if no rejection reaction is registered with the dependent promise, such an unhandled exception will be completely swallowed.

Example 3.1. Docusaurus[3] is an open source application for building, deploying, and maintaining open source websites. It provides a set of scripts for helping users create and manage their websites through running a local server. Figure 1 shows a code fragment from the `start-server.js` file, responsible for starting the Docusaurus web server.

The code fragment is trying to determine whether some selected port on the machine is free and can be bound to. Line 35 calls the `check()` function to determine the availability of the hostname and port and `check()` returns a promise which we will refer to as p. At line 35, the program registers an anonymous function as the fulfill reaction for p. The anonymous function is executed when p is resolved, i.e., the availability of selected hostname and port is checked. If the address is in use, the program prints a descriptive error message and exits, unable to start the server (lines 36–38). If the address is available, the program prints a message to the console, declaring that it is starting the server (line 40). It then instantiates a server object and starts the server locally on the specified port (lines 42–43). The anonymous fulfill reaction ends at line 45 with a running server, ready to support the user's website. Executing this file, the user initially observes the two printed messages regarding availability of the server's address and then starting it. However, according to issue #238[4] of the repository, the server silently fails if there is a mistake in a configuration file required by the server, such as the website's sidebars config file. In this case, the program informs the user of starting the server through the messages, as it normally would. However, the server silently crashes when being instantiated at line 42, without a warning or an indication of failure. Investigating the issue, we find that the exceptions occurred during the server startup are not properly handled. Since the server is instantiated within the anonymous fulfill reaction of promise p, its exceptions only reject the promise created by p`.then()`, and do not surface during the execution. As such, the viewer only sees the printed message indicating a successful server startup, while the server has failed due to the exception. A month after the issue is first submitted to the GitHub repository, a developer finds the root cause of the issue, which as they state: *"makes tracking down bugs in config files super frustrating."* They propose to solve the issue by capturing and handling such exceptions by printing the error along with a stack trace in order to locate the broken configuration. In reality, the submitted fix, depicted in Figure 2, merely adds a `catch` statement that rethrows the exceptions.

[3] https://github.com/facebook/Docusaurus
[4] https://github.com/facebook/Docusaurus/issues/238

```
tcpPortUsed.check(port, 'localhost').then(function(inUse) {
    /*same as before*/
})
+ .catch(function(ex) { setTimeout(function() {throw ex;}, 0); });
```

Fig. 2. The fix made to `start-server.js` by the Docusaurus developers.

```
51 let mysql = require("promise-mysql"), mysqlPool = mysql.createPool({ ... });
52 mysqlPool.getConnection().then(connection => {
53         console.log("Releasing connection...");
54         return connection.release().then(
55             () => console.log("Released successfully!"),
56             () => console.log("Release failed!"));
57     }, err => {
58         console.log("Failed to connect to MySQL " + err); }
59 );
60 mysqlPool.on("release", () => {
61     console.log("The pool reports that release succeeded!");
62 });
```

Fig. 3. JavaScript code fragment from a client of Mysql Promise.

While the fix does not handle the exception in a systematic manner, it causes it to at least be revealed to the user. From the discussion of the pull request, it appears the developers plan to add support for proper handling of the rethrown exception in the future. The pull request is merged nearly two months after the issue is first raised. This particular issue has a simple solution, yet it imposes a severe consequence on the system. Nonetheless, the issue and its accompanying pull request indicate the challenging nature of understanding and debugging promise-based code.

3.2 Unsettled Promises

Every new promise is in the *pending* state, until resolved or rejected. However, not settling a promise results in a *dead* promise, forever pending, preventing the execution of reactions that depend on the promise being settled.

Example 3.2. The Mysql Promise[5] module is a promise-based wrapper for mysqljs,[6] the Node.js client implementation of the MySql protocol. It supports standard database functions for handling connections and connection pools required for communicating with a database server. A pool is a cache of database connections maintained so that the connections can be reused when future requests to the database are required. When done with a pool connection, the programmer must release the connection to return it to the pool, ready to be reused.

The Mysql Promise library is available to developers as an npm module, allowing them to interact with databases through promises. Clients of the Mysql Promise module can create connections and pools, through two functions made available for them, namely `createConnection()` and `createPool()`. Figure 3 displays a typical code snippet of a client of this module. In this example, the developer creates a pool at line 51, which returns a pool object, assigned to `mysqlPool`. At line 52, the developer requests a connection to the `mysqlPool`, which returns a promise. The input to the fulfill reaction of this promise is a `connection`, used for querying the database (line 52).

[5] https://github.com/lukeb-uk/node-promise-mysql
[6] https://github.com/mysqljs/mysql

```
63 poolConnection.prototype.release = function() {
64     - return promiseCallback.apply(this.connection, ['release', arguments]);
65     + this.connection.release();
66 };
67 poolConnection.prototype.destroy = function() {
68     - return promiseCallback.apply(this.connection, ['destroy', arguments]);
69     + this.connection.destroy();
70 };
```

Fig. 4. The fix made to Mysql Promise by its developers.

Once done with the connection, the programmer releases it at line 54, to return it back to the pool. The `connection.release()` returns a promise, which itself has both fulfill and reject reactions registered in the code, at lines 55 and 56, respectively. The programmer expects the `release()` function of the Mysql Promise module to resolve this promise once the connection is successfully released, or reject it otherwise. However, when the programmer executes this code, neither of the two reactions is invoked, meaning neither of the messages at lines 55–56 is printed to the console. According to issue #77[7] of the Mysql Promise module repository, the promises returned by `poolConnection.prototype.release()` are never resolved or rejected by the module. As a result, any client code depending on the fulfillment of `release()` promises is never executed. Trusting the functionality of the module, the developer in our example tries to debug the code, while the code is correct and must function properly according to the documentation of the module. To make matters more confusing, the listener for the `release` event on `mysqlPool` (line 60) gets executed and prints a successful release message(line 61). Meanwhile, the original fulfill reaction at line 55 is never triggered.

When the issue is brought to the attention of the developers of the module, they acknowledge the error and update the implementation of the `release()` function in the module. They directly release the database connections instead of returning unnecessary promises that never settle (Figure 4, line 65). They discover the same problem in their implementation of destroying connections, which is neither reported nor detected prior to this issue (line 69). The library developers fix the issue and publish it to npm in version 3.1.4 of the module. However, as a result, any client programs using these features of the module must update their legacy code to remain consistent with the latest API of the module.

3.3 Implicit Returns in Reactions

Developers often need to execute two or more promises, where execution of each subsequent promise depends on the resolution or rejection of the previous one. To accomplish this, developers use `then()` and `catch()`, returning promises, to create *promise chains*. However, promise chains break silently when the developer forgets to explicitly include a return statement.

Example 3.3. Google Assistant[8] is a virtual personal assistant that helps users find, organize, and get things done in their world. The main way users interact with the Assistant is by carrying on a conversation with it. Apps can extend the Google Assistant by allowing developers to build actions that let users interact with the assistant. Actions on Google[9] client library makes it easy to create apps for the Google Assistant.

[7] https://github.com/lukeb-uk/node-promise-mysql/issues/77
[8] https://assistant.google.com
[9] https://developers.google.com/actions/

```
71  handleRequest (handler) { // class AssistantApp
72      if (typeof handler === 'function') {
73          const promise = handler(this);
74          if (promise instanceof Promise) {
75              promise.then((result) => {
76                  debug(result);
77                  return result;
78              }).catch((reason) => {
79                  this.handleError_('function failed');
80                  this.tell(!reason.message ? ERROR_MESSAGE : reason.message);
81                  return reason;
82              });
83          }
84      }
85  }
```

Fig. 5. Partial JavaScript code segment from the handleRequest() method of the Actions on Google library.

The `AssistantApp` class of this library provides methods for supporting the conversation API protocol of the assistant. The `handleRequest()` method of this class uses a Map of handlers, to asynchronously address incoming Assistant requests (Figure 5). `handleRequest()` takes a handler as an input argument, which can be a function callback or a promise (line 71). After confirming that the type of the input handler is a function at line 72, the program invokes the handler at line 73 and assigns its result to variable `promise`. Next, the program checks the type of the result (line 74), and if it is a promise, registers both fulfill and reject reactions (lines 75 and 78, respectively). If the promise is resolved and the anonymous fulfill reaction at line 75 is invoked, the `result` is returned at line 77. However, the return takes place from within the anonymous fulfill handler of the promise. The result is wrapped in a promise and returned by the `then()` method at line 75, where it is neither returned from the `handleRequest()` function, nor assigned to a variable for future use. Hence, the result of performing the handler function is lost. Similarly, if the promise is rejected, the reject reaction is triggered (line 78). The anonymous reject reaction returns the `reason` of the rejection at line 81, which is again ignored when wrapped in a promise and returned to the `handleRequest()` at line 75. Thus, developers are unable to handle the fulfill and reject reactions of the promises returned by their own handlers.

The discussions revolving around pull request #54[10] of the library's repository, indicate more instances in the code, where the app does not receive the result of handling its request, if the provided handler returns a promise. The fix, depicted in Figure 6, simply returns the promises created based on the outcome of both fulfill and reject reactions, as shown in lines 88 and 95, respectively.

Examples 3.1–3.3 are simplified and summarized instances of a few of the problems developers encounter while taking advantage of the benefits of promises. It is a challenging endeavor to understand the flow of execution and data, and locate the faults in promise-based code. Further, there are more categories of anti-patterns and misuses of promises observed in asynchronous code, as we discuss later in the paper. The prevalence of such issues in real-world applications indicate a need for analyses that can facilitate understanding the semantics of promise-based code.

[10]https://github.com/actions-on-google/actions-on-google-nodejs/pull/54

```
if (promise instanceof Promise) {
    - promise.then(result) => {
    + return promise.then(result) => {
        debug(result);
        return result;
    }).catch((reason) => {
        this.handleError_('function failed');
        this.tell(!reason.message ? ERROR_MESSAGE : reason.message);
        - return reason;
        + return Promise.reject(reason);
    });
}
```

Fig. 6. JavaScript code segment from the Promise-mysql web application.

4 THE PROMISE GRAPH

In this section, we define the notion of a *promise graph*. Our definition is based on the original definition of a promise graph by Madsen et al. [Madsen et al. 2017], but a key difference is that our promise graphs are computed using dynamic analysis, whereas the definition by Madsen et al. is assumed to be computed using static analysis[11]. To emphasize this distinction, we will henceforth use the term *static promise graphs* to refer to the static notion proposed by Madsen et al., and *dynamic promise graphs* to refer to the dynamic notion used in the present paper.

There are several important differences between static and dynamic promise graphs. Dynamic promise graphs reflect promises that are actually created, resolved and rejected during a program execution, whereas static promise graphs reflect all possible executions of a program. Consider an expression that creates a promise (e.g., `Promise.resolve(···)`). If such an expression executes multiple times, a static promise graph would contain a single promise node that abstracts all promises created at that expression. On the other hand, a dynamic promise graph would contain a distinct promise node for each execution of that expression. Furthermore, in a static promise graph, a promise node may have multiple incoming `resolve` and `reject` edges, reflecting the fact that the promises created by a specific expression may be resolved or rejected with different values. By contrast, a dynamic promise graph will reflect that each promise is resolved or rejected exactly once, so it will contain only one incoming edge in such cases.

Beyond the difference between static and dynamic promise graphs, our work extends the notion of promise graphs of Madsen et al. [Madsen et al. 2017] by accommodating all aspects of ECMAScript 6 promises,[12] whereas Madsen et al. only consider a small, idealized subset of promises as embodied in their λ_p calculus. Concretely, some of the key aspects of promises that this paper covers for the first time are: default reactions associated with `then` and `catch`, exceptions, and synchronization constructs such as `all` and `race`. Our analysis also supports linked promises that arise during promise creation, which is not covered in the semantics by Madsen et al.

4.1 Nodes and Edges

The promise graph contains the following nodes:

[11] Note that Madsen et al. do not present any technique for *computing* promise graphs; all the promise graphs shown in their paper were constructed manually.
[12] http://www.ecma-international.org/ecma-262/6.0/#sec-promise-objects

- A *promise node* (p) for each execution of a promise allocation site p in the program. The following idioms give rise to promise nodes: (i) calls to the `Promise` constructor, (ii) calls to the functions `Promise.resolve` and `Promise.reject`, and (iii) calls to methods `then`, `catch`, `all` and `race`.
- A *value node* (v) for each value v that is used to resolve or reject a promise. Note that such values may be lambdas or named functions. Separate value nodes are created for multiple usages of an object, preserving its contextual information at the time is is used to settle a promise.
- A *function node* (f) for every lambda or named function in the program that is registered as a reaction for a promise. The same node is utilized if the function is registered as a reaction for multiple promises.
- A *synchronization node* (s) for each synchronization operation s, where s is an expression consisting of a call `Promise.all(`$[p_1, \cdots, p_n]$`)` or `Promise.race(`$[p_1, \cdots, p_n]$`)`. The purpose of synchronization nodes is to visualize how the state of the promise created by `all` or `race` depends on the state of their input promises p_i, and the order in which they are fulfilled and/or rejected.

The promise graph contains the following edges:

- A *resolve* or *reject* edge $(v) \xrightarrow{\ell} (p)$ from a value node (v) to a promise node (p), where ℓ is a label "resolve" or "reject", reflecting that value v is used to resolve or reject the promise p, respectively.
- A *registration* edge $(p) \xrightarrow{\ell} (f)$, where ℓ is a label "onResolve" or "onReject", from a promise node (p) to a function node (f) represents that the function f is registered as resolve or reject reaction on the promise p, respectively. In other words, it represents the fact that, when the promise p is resolved or rejected, the execution of f is triggered.
- A *link* edge $(p_1) \rightarrow (p_2)$ from a promise node (p_1) to a dependent promise node (p_2) represents the dependency that when a parent promise p_1 is resolved (or rejected) the dependent promise p_2 will be resolved (or rejected) with the same value. Such link edges arise in situations where reactions return values that are promises, or when promises are resolved with values that are promises (see Section 2).
- A *return* or *throw* edge $(f) \xrightarrow{\ell} (v)$, where ℓ is a label "return" or "throw", reflects situations where a function f returns or throws the value v, respectively. Return edges have an optional label "implicit", which is present if a function returns implicitly, i.e., when the programmer has not provided an explicit `return` statement. In such cases, a function returns the value `undefined`.
- We distinguish two kinds of *synchronization edges*, which are associated with the use of `Promise.all` and `Promise.race`. In particular, for an expression $s \equiv p_0$`.all(`$[p_1, \cdots, p_n]$`)` or $s \equiv p_0$`.race(`$[p_1, \cdots, p_n]$`)`, a synchronization edge $(p_i) \xrightarrow{\ell} (s)$ is created that connects the promise node associated with each input promise p_i to the synchronization node (s), where ℓ is a label "resolved", "rejected", or "pending", that reflects whether the input promise p_i was resolved, rejected, or remained unfulfilled, respectively[13]. Furthermore, a synchronization edge $(s) \xrightarrow{\ell} (p)$ connects the synchronization node (s) to the promise node (p), returned by `all` or `race`. Here, the label ℓ is a label "resolved", "rejected", or "pending", reflecting the state of the resulting promise.

[13] It is also possible to provide values instead of input promises in calls to `all` or `race`. In such cases, the label "value" is used.

4.2 Default Reactions

As discussed in Section 2, several situations exist where *default reactions* arise. In our experience, it is very important for developers to be aware of default reactions when trying to understand the behavior of a promise-based program. Therefore, our approach identifies all implicit default reactions and explicitly represents them as function nodes with a special label "missing default reaction". In addition, the promise graphs contains return edges and throw edges reflecting the execution behavior of these default reactions. For example, a call to `then` of the form:

`p.then(`f`)`

is equivalent to:

`p.then(`f`, function(e){ throw e; })`

with a default reject reaction `function(e){ throw e; }`. If this default reject reaction is executed, the promise graph will contain a throw-edge that connects the corresponding `e` to the node corresponding to the promise created by the call to `then`.

5 UNDERSTANDING AND DEBUGGING WITH PROMISE GRAPHS

Our goal is to facilitate understanding and debugging of promise-based code, such as the code shown in Figures 1–6. To this end, we analyze the promise graph and identify a number of anti-patterns that correspond to programming errors. We then visualize the promise graph, overlaid with the findings of our anti-pattern analysis, to further assist developers with their program understanding and debugging tasks. Through the visual graphs, developers can quickly gain an overall understanding of the flow of execution and identify potentially problematic usages of promises.

5.1 Inferring Promise Anti-Patterns

Our work on detecting anti-patterns in promise-based code builds on the work by [Madsen et al. 2017], who defined a semantics of JavaScript promises and proposed the promise graph as a method for understanding and debugging of promise-based code. However, Madsen et al. only constructed promise graphs manually for small snippets of promise-based code, they did not design or implement any analysis for constructing promise graphs and inferring anti-patterns, and they did not report on any experiments that involved the construction of promise graphs for real-world applications. Our work goes beyond the work by Madsen et al. by extending the promise graph to all aspects of JavaScript promises, by identifying several additional anti-patterns, by constructing a dynamic analysis that computes promise graphs, and by reporting on experiments in which promise graphs are constructed for real-world applications.

In this section, we present the anti-patterns detected by our analysis and describe how the collected information is visualized in a way that facilitates program understanding and debugging.

Missing reject reactions. As previously shown in example 3.1, an exception thrown within a fulfill reaction, a reject reaction, or in a function passed to `new Promise()` merely causes the associated promise to be rejected, and does not necessarily manifest itself with an error message that is presented to the user. In other words, any raised exceptions are silently ignored, unless they are explicitly handled in a subsequent reject reaction.

To diagnose unhandled promise rejections, our algorithm analyzes all promise nodes in the graph. If there is no outgoing `onReject` edge from a promise node p (except for the `default` reject reaction), the algorithm adds p to a list L of promises with potentially unhandled rejections. The definition of the missing reject reaction bug pattern of Madsen et al. [2017] would consider all members of L as instances of the missing reject reaction anti-pattern.

However, we can observe that the rejection of p will be handled if: (i) there is a promise p' in the chain, which has registered a reject reaction (an outgoing `onReject` edge to a reaction other than the `default`), and (ii) there is a directed path in the graph from p to p'. In this case, the rejection of p is handled by p''s reject reaction. We apply this heuristic to our algorithm to avoid spurious warnings. Therefore, our analysis only reports missing reject reactions that are not handled further down the promise chain. In such cases, our approach creates a warning note identifying this anti-pattern, and attaches it to the promise node p.

Attempting to settle a promise multiple times. Once a promise is resolved or rejected, its state cannot be changed. Subsequent attempts to settle a promise have no effect on the state of the promise and are ignored, without raising an error.[14] However, such an attempt may reflect a situation where a developer does not understand the semantics of promises. Our analysis detects this anti-pattern by examining the incoming edges to each promise node p. If more than one `resolve` or `reject` edge enters p, the analysis uses the time-stamps associated with these edges to identify the call that settled p. The edges associated with the attempts to re-settle the promise are marked with warning notes indicating the issue.

Unsettled promises.

In some cases, the settling of promises may be contingent on external events (e.g., data arriving at sockets) and an unsettled promise is not necessarily an indication of an error. However, in other cases a promise may remain pending because of a programmer error or if program execution follows an unexpected path, as previously shown in example 3.2. An important consequence of unsettled promises is that their fulfill or reject reactions will never be executed. Our approach detects this anti-pattern by examining all promise nodes. The `state` attribute of each promise node records its state during execution, with all promises initially in the `pending` state, until settled. As our algorithm evolves the graph by analyzing the trace, the `state` attribute of the nodes changes to either `fulfilled` or `rejected` if the promise is settled. When graph creation is complete, the algorithm inspects the graph for nodes that are in the `pending` state, and attaches a warning note to such nodes indicating that the promise is unsettled.

Unreachable reactions. Related to the previous anti-pattern, an unsettled promise p may have registered fulfill and reject reactions that will not be executed. Further, an unresolved promise can be linked to other promises, used to settle promises, or used as input to synchronization operations such as `all()` and `race()`, and thus, prevent their fulfillment as well. To alert programmers at such unreachable reactions, our analysis locates all unexecuted reactions. All function nodes are in the `unexecuted` mode when instantiated. As the program runs, the status of each invoked function is changed to `executed`. When the program execution terminates, the algorithm traverses the graph. All unexecuted functions nodes are marked as unreachable and are added to a list L. However, a subset of these functions may not be executed due to the constraints imposed by the semantics of promises: the execution of fulfill and reject reactions, registered to a promise through `then()`, is mutually exclusive. To account for this, we devise another heuristic that eliminates infeasible reactions from L. In particular, an unexecuted fulfill/reject reaction is no longer considered unreachable if its corresponding reject/fulfill reaction is invoked. Warning notes are attached to the remaining unreachable nodes in L to indicate the anti-pattern.

Note that, as a dynamic analysis, our analysis for constructing promise graphs only reflects observed execution behavior. In general, the fact that promises remain pending or that reactions remain unexecuted may be an artifact of the particular execution that was observed.

[14] Attempting to resolve or reject an already-settled promise is a valid (no-op) operation in JavaScript.

Therefore, warnings of these types produced by our analysis should be studied carefully to determine if a bug exists.

Implicit Returns in Reactions. When a developer forgets to include a `return` statement in a function, the function will implicitly return the value `undefined` when it finishes executing. Recall example 3.3, where the developer forgot to return the promise created by `then()` at line 75 of Figure 5. The newly-created promise was lost, since it was neither assigned to a variable, nor returned from the function. Thus, the function implicitly returned `undefined`, and this value was used to resolve the resulting promise with. As seen in that example, this anti-pattern can entail severe consequences.

To locate instances of this anti-pattern in the graph, we search the graph for function nodes corresponding to functions that returned implicitly. We could mark all such reaction functions as suspicious. However, such an approach would lead to false positives in cases where the promise is located at the end of a chain, in which case its outcome is not used. Therefore, we employ another heuristic that eliminates cases where the return value is not used by another node in the graph (i.e., a reaction function or a linked promise). Our algorithm determines this by inspecting the outgoing edges of the target promise p. If p contains at least one outgoing edge that may use and propagate the implicitly returned `undefined` value, a warning note is attached.

Unnecessary Promises. This situation is also known as the "explicit construction anti-pattern". Unnecessary promises are one of the most common anti-patterns. They usually occur when promises are used as a callback utility without a thorough understanding of their mechanisms. We detect this anti-pattern by searching the graph for pairs of promises, p_1 and p_2, which are connected by a link edge $p_1 \rightarrow p_2$. This indicates that p_2 settles when p_1 does, and with the same value or reason. However, if p_1 is immediately settled through `Promise.resolve()` or `Promise.reject()`, then its creation has been futile. Thus, if the node settling p_1 is a value node with no incoming edges (which indicates an immediate settlement), we report p_1 as an unnecessary promise because, in such cases, the value or the reason used for settling p_1 could have been used to settle p_2 directly.

5.2 Visual Graphs

In the last step of our approach, we create a visual representation of the graphs to assist developers in understanding the behavior of promise-based code. Node-link diagrams are the most common way of displaying graphs [Battista et al. 1998]. We devise a set of visual cues in the diagrams to facilitate understanding of promises, their interactions, and the common anti-patterns. We design a *directed* graph to manifest control- and data-flow dependencies. We lay out the graph in an *axes-oriented* manner, to partially preserve the temporal order of execution, from top to bottom. The graphs may contain cycles and thus we may not be able to present all graphs in a topologically sorted manner. However, most edges are depicted in the same direction, which enables us to effectively visualize hierarchical and temporal relationships, such as those of synchronization nodes. An example of a visualized promise graph is displayed in Figure 7.

We encode a set of visual cues in the graphs in terms of attributes of nodes and edges. For each **node**, we utilize its shape, color, label, and graphics style to convey its characteristics.

- *Shape.* The shape of a node represents its type. We choose ellipses, rectangles, egg shapes, and triangles for promises, functions, values, and synchronization nodes, respectively.
- *color.* We use the *fill* color to indicate the state of execution of the node. The fill color is grey if the node is not executed, orange if the executed node potentially causes a bug, and white

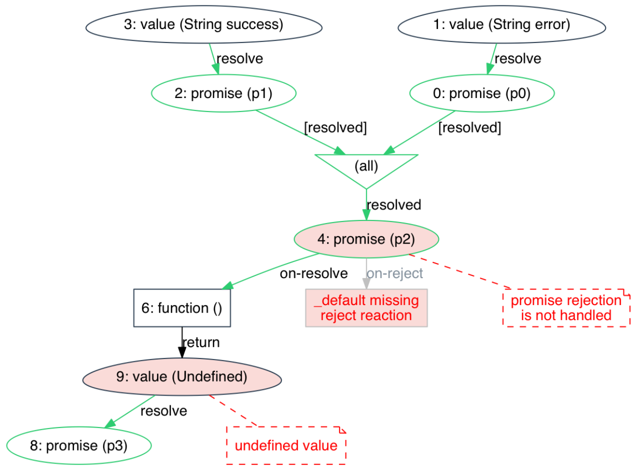


Fig. 7. A visualized promise graph.

otherwise. The *border* color depicts the state of promise nodes, where green, orange, and grey borders indicate fulfilled, rejected, and pending promises, respectively.

- *Style.* The state of the execution of a node is further conveyed with its style, where executed nodes have a solid style, and unreached nodes are grayscaled.
- *Label.* Node labels are designed to display the ID, content, or name of a promise, value, or function node, respectively.

We take advantage of colors, graphics styles, and labels, to represent **edge** attributes.

- *color.* The color of incoming and outgoing edges to promise and synchronization nodes are determined by the status of the node. Consistent with the border color of the node, the edges may be green, orange, or grey. Moreover, a red edge indicates an error in the edge, e.g., an attempt to settle a promise more than once.
- *Style.* The solid style of an edge means it is executed (whether it is used for resolution or rejection, which is depicted by its color). A dashed style of a node's edges indicate the pending state of the node, and thus its reaction edges.
- *Label.* The edge label represents the type of the edge, and varies depending on the source and target nodes. Further, if the edge is faulty (e.g., double settle), the error is manifested in the label of the edge.

Warning Notes. To further assist developers with understanding potential issues with promise-based code, we attach auxiliary nodes, in form of notes, to problematic nodes. The notes are easily distinguishable from regular nodes, by their shapes and the dashed style of their respective links. Further, their red color not only sets them apart from the rest of the nodes, but it also immediately draws the attention of the viewer to these warning signs that need further inspection. The label of a warning note explains the warning, and its link locates the responsible node.

When adjusting our graph visualization framework, we took care to make nodes and links easy to read and track visually. To this end, the drawn graphs have a small number of edge crossings and bends across edges. They have small but uniform edge lengths, with minimal variance. We also prohibit node overlaps, utilize the area by adjusting node sizes based on their labels, and maximize path continuity.

6 IMPLEMENTATION

Our analysis is implemented using the Jalangi instrumentation framework [Sen et al. 2013] in a tool called PromiseKeeper. Jalangi defines callbacks that are invoked at various instrumentation points during program execution, at each such point enabling a client analysis to inspect or modify the application's state. PromiseKeeper's analysis only requires access to the program state at a limited number of instrumentation points: upon entry and exit to functions, before and after function calls, when `return` and `throw` statements are executed, when variable declarations are executed, and when the execution of a new file begins or ends. Below, we summarize the key steps taken by our instrumentation.

Promise creation. When a call `new Promise(···)`, `Promise.resolve` or `Promise.reject` is encountered, a unique identifier p is associated with the newly created promise and a promise node is added to the promise graph to represent promise p. For calls of the form `new Promise(···)`, we additionally record the names of the parameters that are bound to the functions for resolving and rejecting this promise[15] and we update a mapping that records, for each such resolve/reject function, the promise that it is associated with.

Explicitly resolving and rejecting promises. When a call to a resolve or reject function is encountered during execution, we create a value node for the value v that is passed as an argument, use the mapping that was established during promise creation to identify the promise p being resolved or rejected, and add a resolve or reject edge that connects v with p.

Calls to `then`. When a call `p.then(`f_1`,`f_2`)` is encountered, we (i) create function nodes for f_1 and f_2 and connect these nodes with registration edges to the promise p that is bound to the receiver variable `p`[16], (ii) associate a unique identifier with the promise p' that is created by the call to `then` and (iii) add a promise node for p' to the graph. If f_1 is not a function, it is replaced with a default reaction `function(v){ return v; }` and we keep track of this fact so that the node can be highlighted in the graph. Similarly, if f_2 is not a function[17], it is replaced with a default function `function(v){ return v; }`. In all cases, functions used as reactions are wrapped in another function that includes the code for adding nodes and edges to the promise graph.

Calls to `catch`. When a call `p.catch(`f`)` is encountered, we create a function node for f and connect it with a registration edge to the promise p that is bound to the receiver variable `p`, (ii) associate a unique identifier with the promise p' that is created by the call to `catch` and (iii) add a promise node for p' to the graph. Similar to the case for `then`, f is replaced with a default reject reaction `function(e){ throw e; }` if it is not a function, a function node and a registration edge are created for a default resolve reaction `function(v){ return v; }` that is associated with the `catch`, and all these reactions are wrapped in a function that includes the code for adding nodes and edges to the graph. Since calls to `catch` can only specify a reject reaction for a promise, we rewrite such calls into calls to `then` to allow instrumentation of the (default) resolve reaction.

Executing reactions. When execution exits a function f that was registered as a reaction that creates a promise p, there are several cases that we need to consider:

[15] It is customary, but not required, to use the names `resolve` and `reject` for these parameters

[16] Jalangi provides a mechanism that associated a unique object identifier (oid) with each object that is being created. Our instrumentation maintains a mapping between Jalangi's oids and the unique identifiers that we associate with promises. This mapping is consulted to identify the promise p on which `then` is invoked.

[17] This includes the case where `then` is invoked with only one argument, in which case f_2 has the value `undefined`.

(1) If f exited due to the execution of a statement `return` v, there are two cases. If the returned value is a promise p', we create a link edge that connects p' to p. Otherwise, we create a value node for v, a return edge from f to v, and a resolve edge from v to p.
(2) If f exited due to the execution of a statement `throw` e, we create a value node for e, a throw edge from f to e, and a reject edge from e to p.
(3) If f exited without encountering a `throw` or `return` statement, the function implicitly returns the value `undefined`. In such cases, we create an edge from f to a value node for the value `undefined`, and a resolve edge from that node to p. Furthermore, we record the fact that this edge is due to an implicit return, so that this fact can be highlighted in the visual representation of the promise graph.

Calls to `all` and `race`. When a call to `Promise.all(iterable)` or `Promise.race(iterable)` occurs, we create a synchronization node s. We then examine each element of the iterable object that is passed as an argument to `all` or `race`. If the element is a promise, we locate its respective node in the graph and create a `pending` edge from that promise to s. Otherwise, we create a value node representing the element and connect it to s with a resolved edge, reflecting the fact that `all` and `race` treat objects as instantly-fulfilled promises. Both `all` and `race` return a promise p, which is initially in the pending state. A synchronization edge from s to p is added to the graph. The promise p returned by `all/race` is settled as follows: In the case of `all`, p is fulfilled when all input promises are fulfilled, and rejected when one of the input promises is rejected. In the case of `race`, p is fulfilled/rejected as soon as one of the input promises is fulfilled/rejected.

Our implementation supports a number of frameworks and practices commonly utilized by promise-based projects in practice. In addition to native JavaScript promises, we support promises created by bluebird,[18] a promise library extensively used by web developers. We further extended our implementation to include more advanced and custom usages of promises through mechanisms such as proxies.

7 EVALUATION

In this section, we apply PromiseKeeper to a collection of open source JavaScript applications from GitHub to investigate the following research questions:

RQ1 (Characteristics): What is the size of the promise graphs? How many promises are created? How many of those are fulfilled? How many are rejected? How many are never settled?
RQ2 (Performance) What is the performance overhead of PromiseKeeper?
RQ3 (Anti-Patterns): How many instances of promise anti-patterns does PromiseKeeper detect? How much do the proposed heuristics help reduce the number of reported false positives?
RQ4 (Debugging): Are the recorded promise graphs useful for understanding and debugging promise-based applications?

7.1 Experimental Design

To answer these research questions, we run PromiseKeeper on 12 open source promise-based Node.js applications from GitHub. We record the promise graphs, examine their characteristics, and inspect any reported promise anti-patterns.

Experimental Subjects. We selected 12 promise-based Node.js applications from GitHub by searching for JavaScript projects that rely on promises. We selected actively developed/maintained projects that used promises considerably in their logic. A key search criterion was that the projects had a

[18] http://bluebirdjs.com/

Table 1. Summary of characteristic of promise graphs and runtime overhead of PromiseKeeper.

Application			Graph Nodes		Promises			Performance			
Name	**LOC**	**Tests**	**Functions**	**Values**	**Total**	**Fulfilled**	**Rejected**	**Original Runtime (ms)**	**Instr. Runtime (ms)**	**Slowdown Factor**	**Analysis Time (ms)**
1. Actions on Google	16466	25	52	44	44	31	13	560	16355	29.2X	4839
2. Feathers Express	4203	26	1570	577	1008	939	69	320	11280	35.3X	8539
3. Loader Runner	964	26	16	10	12	12	0	344	8485	24.7X	5154
4. Node Fetch	3682	159	846	542	1612	455	114	2230	102107	45.8X	5433
5. Node HN API	6330	10	224	210	2224	224	0	4965	70879	14.3X	5482
6. Promise MySQL	912	6	7	7	21	7	0	1110	7030	6.3X	5007
7. Node Rollout	869	16	57	88	468	88	0	294	10595	36X	18670
8. Node Serialport	11794	167	402	1048	2133	976	75	402	23959	59.6X	22186
9. Promise Branch	197	8	44	38	48	32	10	187	8093	43.3X	6556
10. Razorpay Node.js	7643	157	8	158	746	154	2	525	22376	42.6X	5593
11. Telegraf	4859	88	1124	861	1428	1400	18	1002	8714	8.7X	26710
12. Telegram Mobile	56926	2	0	2	22	2	0	375	17371	47.4X	5300

test suite available. The selected projects include *Actions on Google*, a library for interacting with the Google Assistant, *Node Fetch*, a library that brings the fetch API to Node.js, and *Telegraf*, a Telegram bot framework.

Experimental Setup. We installed each project and its dependencies locally. We then ran the test suite to ensure that our local environment was correctly configured. Next, we ran PromiseKeeper on the application and its test suite to record the promise graphs. With this setup, each unit test gave rise to one execution and one promise graph. In some cases, we had to perform minor adjustments to the test suites to make them compatible with our infrastructure.

We now address each research question in turn and discuss the results.

7.2 RQ1: Characteristics of Promise Graphs

Table 1 shows the results of running PromiseKeeper on each of the 12 applications. The first three columns of the table show the name of the application, the number of lines of source code (excluding code in dependent modules), and the number of tests. The next three columns provide statistics about the promise graphs and report: the number of function nodes, the number of value nodes, and the total number of promise nodes. Then, the columns labeled "Fulfilled" and "Rejected" show how many of the created promises were fulfilled or rejected, respectively. The final four columns report on the run-time overhead of the tool and will be discussed in Section 7.3.

As an example, the "Feathers Express" application consists of 4203 lines of code without considering the project dependencies, and has 26 unit tests. Executing its test suite constructs promise graphs that contained a total of 1570 function nodes, 577 value nodes, and 1008 promise nodes. Of these 1008 promises, 939 were fulfilled and 69 were rejected. In other words, every single promise was either fulfilled or rejected and no promise was left in a pending state after completion of the unit tests.

Finding: Many Test Suites Lack Tests for Rejected Promises. From the results in Table 1, it can be seen that the test suites for the five subject applications at rows 3, 5–7, and 12 do not encounter any rejected promises during their execution. However, promise rejection is an important aspect of promise-based code, commonly utilized by developers as a means to recover from unforeseen error conditions. Not testing any scenarios that entail promise rejection reflects significant shortcomings in these test suites.

Table 2. Summary of detected anti-patterns.

Application	Unsettled Promise	Missing Reject		Broken Chain		Unreachable Reaction		Multiple Settle	Unnecessary Promise
		Before	**After**	**Before**	**After**	**Before**	**After**		
1. Actions on Google	0	13	4	12	5	26	6	1	0
2. Feathers Express	0	230	4	10	9	349	12	0	66
3. Loader Runner	0	2	2	12	6	0	0	0	0
4. Node Fetch	1012	341	78	155	150	449	119	0	119
5. Node HN API	0	384	384	29	0	0	0	0	180
6. Node Rollout	380	85	47	3	0	52	52	0	15
7. Node Serialport	2170	410	8	320	183	650	83	0	182
8. Promise Branch	0	11	7	64	0	126	126	0	54
9. Promise MySQL	14	0	0	0	0	0	0	0	0
10. Razorpay Node.js	590	2	0	9	3	4	2	2	0
11. Telegraf	8	386	386	18	0	568	22	48	70
12. Telegram Mobile	20	0	0	0	0	0	0	0	0
Total	4194	1864	920	632	456	2224	185	51	686

7.3 RQ2: Performance Overhead

To assess the performance of PromiseKeeper, we measured the time required to analyze the subject applications. We consider two types of performance overhead: (i) run-time overhead incurred by our instrumentation as it gathers a trace of relevant operations, and (ii) time required for postprocessing traces, generating promise graphs, detecting anti-patterns, and creating and persisting visual graphs and their reports. The last set of columns in Table 1 show: the original time required to run the test suite (column "Original Runtime"), the running time when our instrumentation for creating promise graphs is enabled (column "Instr. Runtime"), the slowdown factor (computed by dividing the latter by the former), and the time required to postprocess the trace gathered by the instrumentation ("Analysis Time"). To compute these numbers, we ran each experiment three times and report the averages.

For example, running the test suite of "Feathers Express" with PromiseKeeper's instrumentation enabled required 11280 milliseconds, which is 35.3X slower than running the original test suite. Furthermore, PromiseKeeper required 8539 milliseconds to postprocess the results and generate the promise graph for this application.

7.4 RQ3: Detecting Anti-Patterns

Table 2 reports on the total number of anti-patterns detected in the promise graph, as discussed in subsection 5.1, and on the effectiveness of the proposed heuristics for filtering spurious warnings. Overall, we found a total of 4194 unsettled promises, 1864 missing reject reactions, 632 broken promise chains, 2224 unreachable reactions, 51 attempts to resettle a settled promise, and 686 unnecessary promises, prior to deploying the heuristics.

Table 3 displays the number of unique static locations with respect to each anti-pattern. For instance, the 1012 instances of unsettled promises observed for the Node Fetch application in row 4 of Table 2 are caused by various executions of only 17 uniques locations in the application. Overall, the detected anti-patterns corresponded to 53 unique locations for unsettled promises, 299 for missing reject reactions, 186 for broken promise chains, 88 for unreachable reactions, 8 for resettling settled promises, and 52 for unnecessary promises. Below, we provide further analysis for each of the anti-patterns.

Unsettled Promises. In the aggregate over all subject applications, 43% of all promises remained pending at the end of execution, which may reflect incompleteness of the test suite. We also observed that many applications use non-native promise libraries like bluebird, which rely on

Table 3. Summary of unique static locations of detected anti-patterns.

Application	Unsettled Promise	Missing Reject		Broken Chain		Unreachable Reaction		Multiple Settle	Unnecessary Promise
		Before	**After**	**Before**	**After**	**Before**	**After**		
1. Actions on Google	0	9	3	9	4	6	6	1	0
2. Feathers Express	0	30	3	10	9	4	4	0	4
3. Loader Runner	0	2	2	4	2	0	0	0	0
4. Node Fetch	17	170	13	117	112	13	13	0	27
5. Node HN API	0	3	3	1	0	0	0	0	1
6. Node Rollout	15	20	11	1	0	8	8	0	1
7. Node Serialport	8	50	5	33	15	47	47	0	10
8. Promise Branch	0	11	6	8	0	5	5	0	5
9. Promise MySQL	4	0	0	0	0	0	0	0	0
10. Razorpay Node.js	2	1	0	2	1	2	2	2	0
11. Telegraf	4	3	3	1	0	3	3	5	4
12. Telegram Mobile	3	0	0	0	0	0	0	0	0
Total	53	299	52	186	143	88	88	8	52

unsettled promises as the basis for some optimizations. As such, the 4194 unsettled promises were created at only 53 unique locations in code. We conclude that the detection of unsettled promises can be helpful for developers to assess the quality and completeness of their test suite.

Missing Reject Reactions. We discovered a total of 1864 missing reject reactions across all applications, as shown in the "before" subcolumn of the "Missing Reject" column of Table 2. However, a missing-reject warning can be viewed as a false positive if promise rejection is handled downstream in a promise chain. As the "after" subcolumn of the "Missing Reject" column shows, applying this heuristic reduces the number of missing reject reactions to 920 across all subject applications, rooted in only 299 unique locations in code.

Furthermore, we adjusted our analysis to report those cases where promises with unhandled rejections were rejected in practice and found seven such cases in "Node Fetch". Such cases are easily overlooked due to missing error-handling code and incompleteness of test suites. However, they can indicate real bugs and thus PromiseKeeper provides a mechanism for detecting them.

Implicit Returns in Reactions. This anti-pattern detects situations where a function that is registered as a reaction does not explicitly return a value. This scenario may arise when the programmer accidentally forgets to include a `return` as in Figure 6. Note that, in the case where the value that should have been returned is a promise, the lack of an explicit return may give rise to a broken promise chain. Overall, we found 632 such cases, induced by 186 unique locations, across all subject applications. However, the lack of a return is harmless if the reaction occurs at the end of a promise chain, when there is no subsequent reaction that uses the value. Applying our heuristic reduces the number of issues to 456, in 143 static locations.

Unreachable Reactions. The "before" subcolumn of the "Unreachable Reaction" column depicts the number of function nodes in the promise graphs that are not executed. The results show a total of 2224 unreachable function nodes observed across all applications, caused by multiple executions of only 88 unique statement in code.

However, as explained in subsection 5.1, since a promise is settled only once, at most one of its fulfill and reject reactions will be executed. Therefore, we can remove from the initial set of unreachable reactions any function node for which the "opposite" reaction has been executed. As can be seen in Table 2, this heuristic reduces the number of unreachable function nodes to just 422.

Multiple Attempts to Settle a Promise. We found 51 instances of this anti-pattern, which attributed to only eight unique locations. 48 of the detected instances belonged to the "Telegraf" framework

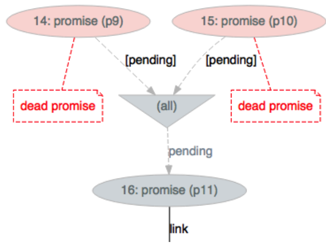


Fig. 8. Example of unsettled promise, Actions on Google.

```
98  it('allow cloning a response, use both as stream'
99    , function () { var url = base + 'hello';
100   return (0, _src2.default)(url).then(function (res) {
101     var r1 = res.clone();
102     var handler = function dataHandler(chunk) {
103       expect(chunk.toString()).to.equal('world'); };
104     return Promise.all([streamToPromise(res.body,
105       handler), streamToPromise(r1.body, handler)]);
106   });
107 });
```

Fig. 9. [Simplified] test from Node Fetch.

(row 11). We are not aware of any reason for attempting to settle an already-settled promise and were surprised to find 51 instances of this anti-pattern by only exercising the application's test suites.

Unnecessary Promises. The creation of unnecessary promises is a prevalent anti-pattern and PromiseKeeper found 686 instances of it across all subject applications, which originated in 52 unique locations. This anti-pattern, also known as the *explicit construction anti-pattern*, is recognized by the official bluebird documentation as the most common anti-pattern.[19]

7.5 RQ4: Debugging with Promise Graphs

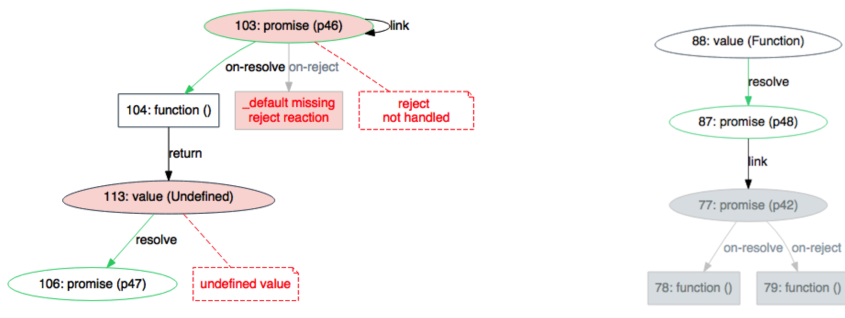
In this section, we discuss five examples of issues found by our tool, depicted in Figures 8–11b, and explain how PromiseKeeper assists developers with understanding and locating these anti-patterns, through reports and visualized graphs.

Example 7.1. Unsettled Promises. Figure 9 displays a simplified unit test from the test suite of Node Fetch.[20] This test aims at cloning an XMLHttpRequest (XHR) response, and promisifying both streams of original and cloned versions of the incoming response. The test starts with creating a URL (line 99) that is used for sending an XHR request at line 100. Next, the test creates a clone for the response of the XHR request, `res`, at line 101. Finally, at line 105, the test creates two promises for both response streams through the `streamToPromise()` method. This method returns a new promise for each stream, which is resolved when the handling of the data steam terminates. At line 105, the test takes advantage of `Promise.all()`, to return when both promises settle. However, the report of exercising this test with PromiseKeeper indicates that neither of these promises settles.

The graph generated by PromiseKeeper for this test is (partially) displayed in Figure 8. Nodes 14 and 15 of the graph represent the promises created for the two response streams, namely `p9` and `p10`. As displayed in the graph, neither of these promises have settled, and thus promise `p11`, created by`Promise.all()`, is pending as well. This part of the graph is greyscaled as it has not been dynamically exercised. As such, all unexecuted reactions of unsettled promises are counted towards unreachable reactions, reported in the "Unreachable Reaction" column of Table 1.

Example 7.2. Missing Reject Reactions. PromiseKeeper's reports contain the information of missing reject reactions, before and after applying the heuristic. However, we only mark the refined set of missing reject reactions on the graph, to avoid overwhelming the viewer. An example of how the promise graph guides developers toward detecting this anti-pattern is shown in Figure 10a, a

[19] http://bluebirdjs.com/docs/anti-patterns.html
[20] https://github.com/bitinn/node-fetch



(a)

(b)

Fig. 10. Examples of missing reject reaction, Telegraf (10a) & unreachable reaction, Serialport (10b).

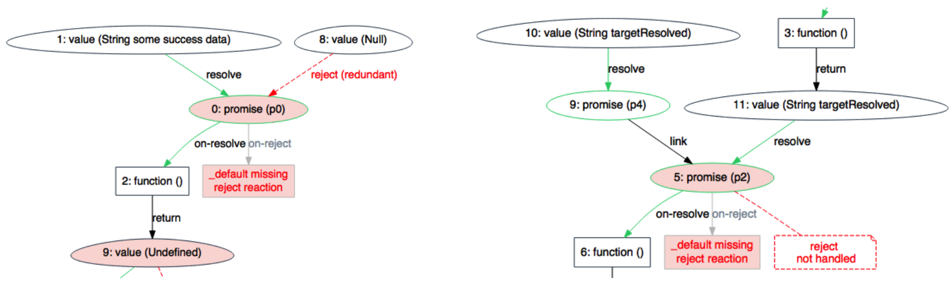
part of a graph generated for the test suite of Telegraf.[21] It can be seen on the graph that node #103, representing promise p13, is marked with red. The attached warning note identifies the issue as the missing reject reaction anti-pattern. The graph and its accompanying report can be used to recognize and locate the issue, which can then be solved by registering a reject reaction to p13.

Example 7.3. Unreachable Reactions. Greyscaled nodes and dashed edges enable the developers to locate unreachable nodes and unexercised reactions in a glance. A promise graph inferred from Node Serialport,[22] partially shown in Figure 10b, entails an unexecuted (pending) promise (node #77) and consequently, two unreachable reaction functions (nodes #78 and #79). Having the graph, we can see that the pending promise p42 is linked to another promise p48, and based on semantics of promises, we expect them to have the same state of settlement. However, we notice that p48 is fulfilled, while p42 remains pending, indicating an error. Inspecting the code, we locate the root cause of the problem, function (#88) used for resolving p48, which is an asynchronous callback, preventing the settlement of p42.

Example 7.4. Multiple Attempts to Settle A Promise. Figure 11a shows a situation where an attempt is made to re-settle a promise. The problematic operation, the `reject` edge from value node #8 to promise `p0` (node #0), manifests the anti-pattern utilizing its style, color, and label "redundant". Here, the program attempted to reject `p0` with a Null value but `p0` was already fulfilled by value node #1. We discuss a common practice for remedying multiple settles in subsection 7.6.

Example 7.5. Unnecessary Promises. The graph shown in Figure 11b illustrates (part of) the execution of a unit test from Promise Branch.[23] Consider node #5, displaying promise p2, which is linked to and resolved by promise p4 (node #9). However, p4 itself is immediately resolved by a primitive value, i.e., the "targetResolved" string (value node #10). Examining the promise graph, we suspect the reason for creation of p4, as it serves no purpose in the graph other than immediately resolving p2. It seems that p2 could have been directly resolved with the value node #10, without requiring to create another promise. Inspecting the code, we finds that a `then()` reaction in the test case (line 108) is indeed returning an unnecessary promise from line 110. We find more instances of unnecessary promises in the application, which complicate code unnecessarily. Eliminating such intermediate promises will not affect program behavior, but it will make the code more comprehensible, maintainable, and efficient.

[21] https://github.com/telegraf/telegraf/
[22] https://github.com/node-serialport/node-serialport
[23] https://github.com/danielglennross/promise-branch



(a) (b)

Fig. 11. Examples of multiple settle, Razorpay (11a), and unnecessary promise, Promise Branch (11b).

```
Promise.resolve().then(function () {
    return branch(Promise.resolve(), function () {
        return Promise.resolve('targetResolved');})});
```

We find more instances of unnecessary promises in the application, which complicate the execution. Eliminating the middle promise will not affect the semantics of the code, but it will make the code more comprehensible and maintainable.

7.6 Common Developer Practices

While consulting online documentation, inspecting GitHub issues, and examining promise-based projects, we noticed a set of common practices that developers use to prevent promise anti-patterns.

Multiple attempts to settle. This anti-pattern may occur when developers devise different scenarios for fulfilling or rejecting a promise. Alternatively, they can design the code such that the methods "race" against each other to resolve a promise. However, extensive use of deferred promises is an anti-pattern itself, as it impedes comprehension of the execution flow.

Unsettled promises. A band-aid solution we observed for this anti-pattern is manually rejecting unsettled promises through a timeout. Obviously, this approach does not locate and resolve the root cause of the issue.

Rethrowing exceptions. To prevent unexpected exceptions from rejecting promises and going unnoticed, many applications re-throw such exceptions, in order to later handle the `uncaughtException` event through either the JavaScript process,[24] or bluebird's error management configuration.[25]

The mere existence of such treatments indicate that developers find the use of promises challenging, and that they have formed common and informal practices to prevent or remedy the pitfalls of using promises. Our observations support the need for facilitating this process through systematic approaches, such as PromiseKeeper.

7.7 Threats to Validity

The representativeness of our subject applications is a threat to the validity of the experiments. We addressed this threat by randomly selecting open-source Node.js applications of various domains and sizes on GitHub. The second threat is the generalizability of the investigated scenarios and the examiners' bias, which we mitigated by using the test suites of the subject applications. Another concern is the impact of the coverage of the test suites on the results of the experiments. Dynamic analysis is incomplete and thus PromiseKeeper may miss anti-patterns in uncovered parts of

[24] https://nodejs.org/api/process.html#process_event_uncaughtexception
[25] http://bluebirdjs.com/docs/api/error-management-configuration.html

the applications. However, PromiseKeeper provides precise information regarding anti-patterns within the application code as well as test code. Our experiments are reproducible since we plan to release our implementation of PromiseKeeper, the experimental subjects, and all reports.

8 RELATED WORK

Promise graphs were originally proposed by Madsen et al. [2017], who also defined the semantics of promises for λ_p, a small calculus that extends the λ_{JS} calculus by Guha et al. [2010] with features for promise creation, registering reactions, and resolving and rejecting promises. As was previously discussed in Section 4, we extend the promise graphs of Madsen et al. to handle the full definition of promises in the ECMAScript 6 specification. Furthermore, Madsen et al. did not present any automated technique for constructing promise graphs as we do in this paper, nor did they report on any experiments with an automated analysis tool such as *PromiseKeeper*.

Several other projects focus on JavaScript promises. Loring et al. [2017] propose λ_{async}, a semantics of the asynchronous execution model of Node.js. Similar to Madsen et al. [2017], they express their model as an extension of the λ_{JS} calculus by Guha et al. [2010]. Loring et al. sketch several applications of their semantics, including race detection, time-travel debugging, and resource and priority scheduling. Kambona et al. [2013] report on a case study that investigates the effectiveness of promises and reactive extensions using an example of an online collaborative drawing editor. Brodu et al. [2015] propose a technique to automatically refactoring legacy JavaScript code, written using event-handling and callbacks, into promise-based code.

Gallaba et al. [2017] present an empirical study in which they report on the prevalence of callback-accepting functions and callsites, and discuss several solutions for rewriting callback-based code using alternative mechanisms for accommodating asynchrony, including Async.js and promises. None of the above works present any automated analysis or tool for detecting errors in promise-based code.

Several other papers target analysis of asynchronous JavaScript execution in different domains. Frameworks such as Arrows [Khoo et al. 2009] have been proposed to facilitate understanding and writing asynchronous code and help developers avoid asynchronous errors. Sahand [Alimadadi et al. 2016a] is a comprehension tool that infers a temporal model of asynchronous interactions in full-stack JavaScript. Tochal [Alimadadi et al. 2015] proposes a hybrid impact analysis technique that includes XHR objects and their interactions with the code in the data- and control-flow analysis. Unlike our work, these papers do not present program understanding tools for JavaScript promises.

Much previous research has been dedicated to dynamic analysis of JavaScript for various different purposes. Sen et al. present Jalangi, an instrumentation framework for dynamic program analysis of JavaScript applications [Sen et al. 2013]. Jalangi has been used as the basis for various tools, such as DLint, a "lint" tool based on dynamic program analysis [Gong et al. 2015], MemInsight, a platform-independent memory debugging tool [Jensen et al. 2015], and Crowdie, a feedback-directed debugging tool [Madsen et al. 2016]. Considerable research has been devoted to tools for detecting and repairing event races in JavaScript applications [Adamsen et al. 2017b,a; Petrov et al. 2012; Raychev et al. 2013; Zhang and Wang 2017; Zheng et al. 2011]. Barr et al. [2016] investigate time-travel debugging for JavaScript, which enables a precise analysis of the history of statements and values that lead to an error. There are numerous dynamic analysis techniques based on collecting and analyzing JavaScript traces selectively [Amalfitano et al. 2014; Hibschman and Zhang 2015]. Some assist with the process of comprehension and debugging by utilizing information visualization techniques and providing higher-level models of execution [Alimadadi et al. 2018, 2016b; Maras et al. 2013; Zaidman et al. 2013]. Andreasen et al. [2017] present a survey of dynamic analysis techniques for JavaScript. However, we are not aware of previous dynamic analysis techniques that target the use of promises.

9 CONCLUSIONS

In previous work, Madsen et al. [2017] formalized the semantics of JavaScript promises and defined the promise graph, a data structure for helping programmers understand and debug promise-based code. However, their notion of the promise graph did not cover all promise-related features in the ECMAScript 6 specification, and they did not propose or evaluate any technique for constructing promise graphs.

In this paper, we have presented a dynamic analysis for constructing promise graphs for all promise-related features in the ECMAScript 6 specification. We also reported on its implementation in *PromiseKeeper*, and on an evaluation in which we apply *PromiseKeeper* to 12 promise-based Node.js applications taken from GitHub. Our findings show that *PromiseKeeper* is capable of constructing promise graphs for large and complex applications with acceptable run-time overhead. Furthermore, we demonstrate the tool's ability to detect anti-patterns such as missing reject reactions, attempts to settle a promise multiple times, unsettled promises, unnecessary promises, and unreachable code that warrant further investigation by a developer. We convey these findings using a visual representation that enables developers to quickly obtain an understanding of the behavior of, and potential problems in promise-based code.

10 ACKNOWLEDGEMENTS

This work was supported in part by NSF grant CCF-1715153 and the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. 2017b. Repairing Event Race Errors by Controlling Nondeterminism. In *Proc. 39th International Conference on Software Engineering (ICSE).*

Christoffer Quist Adamsen, Anders Møller, and Frank Tip. 2017a. Practical Initialization Race Detection for JavaScript Web Applications. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).*

Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2015. Hybrid DOM-Sensitive Change Impact Analysis for JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP).* LIPIcs, 321–345.

Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2016a. Understanding Asynchronous Interactions in Full-Stack JavaScript. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE).* ACM, 1169–1180.

Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2018. Inferring Hierarchical Motifs from Execution Traces. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE).* ACM, 12 pages.

Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. 2016b. Understanding JavaScript Event-Based Interactions with Clematis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 2 (2016), 38.

Domenico Amalfitano, AnnaRita Fasolino, Armando Polcaro, and Porfirio Tramontana. 2014. The DynaRIA tool for the comprehension of Ajax web applications by dynamic analysis. *Innovations in Systems and Software Engineering* 10, 1 (2014), 41–57.

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. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 66.

Earl T. Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. 2016. Time-travel Debugging for JavaScript/Node.Js. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016).* ACM, 1003–1007.

Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. 1998. *Graph drawing: algorithms for the visualization of graphs.* Prentice Hall PTR.

Etienne Brodu, Stéphane Frénot, and Frédéric Oblé. 2015. Toward automatic update from callbacks to Promises. In *Proceedings of the 1st Workshop on All-Web Real-Time Systems.* ACM, 1.

ECMA Ecmascript. 2015. Language Specification. (2015).

Keheliya Gallaba, Quinn Hanam, Ali Mesbah, and Ivan Beschastnikh. 2017. Refactoring Asynchrony in JavaScript. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME).* IEEE Computer Society, 11 pages.

Erich Gamma. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: dynamically checking bad coding practices in JavaScript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 94–105.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The essence of JavaScript. In *European conference on Object-oriented programming*. Springer, 126–150.

Joshua Hibschman and Haoqi Zhang. 2015. Unravel: Rapid Web Application Reverse Engineering via Interaction Recording, Source Tracing, and Library Detection. In *Proceedings of ACM User Interface Software and Technology Symposium (UIST)*. ACM, 270–279.

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*. ACM, 345–356.

Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter. 2013. An evaluation of reactive programming and promises for structuring collaborative web applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. ACM, 3.

Yit Phang Khoo, Michael Hicks, Jeffrey S Foster, and Vibha Sazawal. 2009. Directing JavaScript with arrows. *ACM SIGPLAN Notices* 44, 12 (2009), 49–58.

Matthew C Loring, Mark Marron, and Daan Leijen. 2017. Semantics of asynchronous JavaScript. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 51–62.

Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2017. A Model for Reasoning About JavaScript Promises. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*.

Magnus Madsen, Frank Tip, Esben Andreasen, Koushik Sen, and Anders Møller. 2016. Feedback-directed instrumentation for deployed JavaScript applications. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 899–910.

Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. Static Analysis of Event-Driven Node.js JavaScript Applications. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*.

Josip Maras, Maja Stula, and Jan Carlson. 2013. Generating Feature Usage Scenarios in Client-side Web Applications. In *Proceeding of the International Conference on Web Engineering (ICWE)*. Springer, 186–200.

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

PromiseKeeper 2018. PromiseKeeper. https://github.com/nuprl/PromiseKeeper. (2018).

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

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*. ACM, 488–498.

Andy Zaidman, Nick Matthijssen, Margaret-Anne Storey, and Arie van Deursen. 2013. Understanding Ajax applications by connecting client and server-side execution traces. *Empirical Software Engineering* 18, 2 (2013), 181–218.

Lu Zhang and Chao Wang. 2017. RClassify: Classifying Race Conditions in Web Applications via Deterministic Replay. In *Proc. 39th International Conference on Software Engineering (ICSE)*.

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