Orchestrating Dynamic Analyses of Distributed Processes for Full-Stack JavaScript Programs

Laurent Christophe
Software Languages Lab, Vrije Universiteit Brussel
Brussel, Belgium
lachrist@vub.ac.be

Coen De Roover
Software Languages Lab, Vrije Universiteit Brussel
Brussel, Belgium
cderoove@vub.ac.be

Elisa Gonzalez Boix
Software Languages Lab, Vrije Universiteit Brussel
Brussel, Belgium
egonzale@vub.ac.be

Wolfgang De Meuter
Software Languages Lab, Vrije Universiteit Brussel
Brussel, Belgium
wdmeuter@vub.ac.be

Abstract

Dynamic analyses are commonly implemented by instrumenting the program under analysis. Examples of such analyses for JavaScript range from checkers of user-defined invariants to concolic testers. For a full-stack JavaScript program, these analyses need to reason about the state of the client-side and server-side processes it is comprised of. Lifting a dynamic analysis so that it supports full-stack programs can be challenging. It involves distributed communication to maintain the analysis state across all processes, which has to be deadlock-free. In this paper, we advocate maintaining distributed analysis state in a centralized analysis process instead —which is communicated with from the processes under analysis. The approach is supported by a dynamic analysis platform that provides abstractions for this communication. We evaluate the approach through a case study. We use the platform to build a distributed origin analysis, capable of tracking the expressions from which values originate from across process boundaries, and deploy it on collaborative drawing application. The results show that our approach greatly simplifies the lifting process at the cost of a computational overhead. We deem this overhead acceptable for analyses intended for use at development time.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GPCE '18, November 5–6, 2018, Boston, MA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6045-6/18/11...$15.00
https://doi.org/10.1145/3278122.3278135

CCS Concepts • **Software and its engineering** → **Software maintenance tools**;

Keywords Dynamic Analysis, Distributed Applications, JavaScript

ACM Reference Format:
Laurent Christophe, Coen De Roover, Elisa Gonzalez Boix, and Wolfgang De Meuter. 2018. Orchestrating Dynamic Analyses of Distributed Processes for Full-Stack JavaScript Programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3278122.3278135

1 Introduction

JavaScript has become the de facto distributed programming language for the web. Its nodeJS platform has given rise to so-called "full-stack" JavaScript programs. The client and server processes of these programs are implemented entirely in JavaScript, and communicate through the standardized XMLHttpRequest and WebSocket APIs for requesting and bidirectional messaging respectively. Development tools for understanding (e.g., [1]), maintaining (e.g., [7, 12, 16]), testing (e.g., [5, 10]), and validating (e.g., [8, 15]) JavaScript are following suite. We refer the reader to Andreasen et al. [2] for a comprehensive survey. As JavaScript is notorious for being difficult to analyze statically, most of these tools are founded on dynamic analysis instead. Accordingly, several frameworks (e.g., [9, 19, 22]) have been proposed to facilitate building dynamic analysis tools for single-process JavaScript programs. Unfortunately, these frameworks offer no dedicated support for building analyses that target distributed JavaScript programs.

To analyze a distributed program, one can either analyze each process separately and merge the per-process analysis results post-mortem, or orchestrate processes under analysis to maintain the distributed state of the analysis. Live orchestration is the only viable option for

analyses that need to intervene in the execution of the distributed program under analysis but it is technically challenging. For instance, when invariants such as "a user cannot be logged in via more than one client" need to be enforced across the clients of a stateless (RESTful) server. This would require instrumenting each client process so that the state of the invariant is checked with the other clients upon every request to the server. A technical challenge for the analysis developer to overcome is that the client process should not resume its execution before its request has been green-lighted, but should remain responsive to incoming communication from other clients that need to check the invariant. It is error-prone and a duplication of effort to task every analysis developer with ensuring that this orchestration is deadlock-free.

In this paper, we propose an approach —supported by a dynamic analysis platform with a distinct architecture— that facilitates building dynamic analyses for distributed JavaScript applications. An analysis implemented according to this approach features a centralized analysis process that is shared data from each process under analysis. The analysis developer intervenes in the execution of these processes through *remote references*, which are abstractions for interacting with objects located in remote processes.

The contributions of this paper are three-fold:

- We propose an approach to building dynamic analyses that facilitates supporting the analysis of distributed programs. The approach advocates maintaining distributed analysis state in a centralized analysis process which is communicated with from the processes under analysis. We motivate this architecture through a running example of an analysis that checks a distributed invariant.
- We support this approach by a dynamic analysis platform called ARAN-REMOTE that provides abstractions for the distributed communication that is required to maintain the analysis state. These abstractions comprise remote procedure calls and remote references. To reduce the accidental complexity that comes with their use, they can be made synchronous via a domain-specific communication protocol and therefore isomorphic to regular procedure calls and regular references respectively. The platform is available as open source software[1] and can be deployed on ECMAScript2015-compliant engines, including mainstream browsers and nodeJS.
- We evaluate our approach by following it to implement a dynamic analysis that tracks the expressions values originate from across distributed process boundaries, and by comparing this implementation with one on top of ARAN, a state-of-the-art instrumentation platform for single-process JavaScript programs.

The remainder of this paper is organized as follows. Section 2 uses a motivating example to illustrate the complexity of manually orchestrating dynamic analysis processes to maintain distributed analysis state. Section 3 sketches an overview of our approach and the API of the supporting platform architecture that obviates this burden. Section 4 discusses its implementation for JavaScript on top of the ARAN instrumentation platform. Section 5 evaluates our approach by means of a representative distributed dynamic analysis implemented according to it. Section 6 surveys related work, while Section 7 concludes the paper.

2 Motivating Example

We start our exposition with a motivating example that illustrates the difficulties of lifting a dynamic analysis for single-process programs to distributed ones.

2.1 Invariant Checking of a Single-Process nodeJS Application

Files being written to concurrently are often the cause of bugs in file-manipulating programs. For instance, a log of HTTP requests might become corrupted when two server processes don't concatenate their information to it atomically. Imagine a nodeJS application, `app.js`, for which the developers implemented a mechanism to prevent files from being opened concurrently. To test their mechanism, the developers built a dynamic analysis on top of ARAN-LOCAL[2], a tiny wrapper around ARAN[3] which is a state-of-the-art instrumentation platform for single-process JavaScript programs [9]. The analysis inserts run-time checks into the application that detect violations of the invariant "a file should never be opened twice".

[1] https://github.com/lachrist/aran-remote

[2] https://github.com/lachrist/aran-local

[3] https://github.com/lachrist/aran

```
// aran-local-node --analysis=analysis.js -- app.js
const fs = require("fs");
const acorn = require("acorn");
module.exports = (options, callback) => {
  const fdpaths = () => fs.readdirSync("/proc/self/fd")
    .map((fd) => fs.readlinkSync("/proc/self/fd/"+fd));
  const invoke = (obj, key, args) => {
    if (obj === fs && key === "open")
      if (fdpath().includes(args[0]))
        throw new Error("File already opened");
    return obj[key](...args);
  };
  callback(null, {parse:acorn.parse, advice:{invoke}});
};
```

Listing 1. Analysis for single-process `app.js`

Listing 1 depicts the implementation of the analysis. Like other ARAN-LOCAL analyses, it is implemented by asynchronously returning a *parse function* and an *advice object* (line 13). The parse function indicates which files of the application should be instrumented. When this function returns a falsy value, ARAN-LOCAL will leave the file untouched. Otherwise, the function should return an ECMAScript Tree[4]. The advice object exposes user-defined functions called *traps* which will be called at run time from the instrumented code. For instance, the method invocation `o[k](x1, x2)` could be transformed by ARAN into the expression `ADVICE.invoke(o, k, [x1, x2], 123)`. The last argument passed to the trap function is always an integer that uniquely identifies the corresponding ESTree node in the original file.

For this analysis, the developers wanted to analyze their entire application, so they parsed each of its files using the npm module ACORN (lines 3 and 13). Knowing that their application always manipulates files by first invoking `fs.open`[5], they only had to define a trap for method invocations. The trap tests whether the invocation would lead to a file being opened twice. An exception is thrown when this is the case. Otherwise, the invocation is executed.

2.2 Naive Invariant Checking of a Distributed nodeJS Application

To improve its performance, the developers of our hypothetical application decided to distribute it across two nodeJS processes: `app1.js` and `app2.js`. The application's invariant "a file should never be opened twice" should now be respected across all of the application's processes. To lift their analysis to distribution, the developers have little choice but to deploy an ARAN-LOCAL analysis on each of the processes of the application under analysis —in the remainder of this paper, we will refer to these processes as *target processes*— and to perform inter-process communication. They decide to carry out this communication using HTTP requests and WebSocket messaging. These technologies require a standard client-server model. Listing 2, the client, is an analysis instance which is deployed for each of the target processes. Listing 3, the server, handles both HTTP requests and WebSocket connections originating from the analysis instances.

```
// aran-local-node --host=localhost:8000
//                 --analysis=client-analysis.js
//                 -- app1.js
const fs = require("fs");
const acorn = require("acorn");
const XMLHttpRequest = require("xmlhttprequest").XMLHttpRequest;
const WebSocket = require("ws");
module.exports = ({argm:{host}}, callback) => {
  const fdpaths = () => fs.readdirSync("/proc/self/fd")
    .map((fd) => fs.readlinkSync("/proc/self/fd/"+fd));
  const websocket = new WebSocket("ws://"+host);
  websocket.onmessage = () => { websocket.send(fdpaths()) };
  const invoke = (obj, key, args) => {
    if (obj === fs && key === "open") {
      const request = new XMLHttpRequest();
      request.open("GET", "http://"+host+"/"+args[0], false);
      request.send();
      if (request.status === 403)
        throw new Error("File already opened");
    }
    return obj[key](...args);
  };
  websocket.onopen = () => {
    callback(null, {parse:acorn.parse, advice:{invoke}});
  };
};
```

Listing 2. Client-side analysis for distributed `app.js`

Aside from its use of the npm modules XMLHTTPREQUEST and WS, Listing 2 is similar to Listing 1. Line 11 uses WS to establish a WebSocket connection over which the server will push requests for information about the files that are currently open by the client process. Lines 15, 16, and 17 use XMLHTTPREQUEST to perform a synchronous HTTP request to the server for its approval of the opening of the file. Note that this HTTP request has to be performed synchronously as its result is required to decide whether to continue with the invocation or to throw an error.

[4] https://github.com/estree/estree
[5] https://nodejs.org/api/fs.html#fs_fs_open_path_flags_mode_callback

```
1  // node server-analysis.js 8000
2  const wss = new require("ws").Server({
3    noServer: true,
4    clientTracking: true
5  });
6  const server = require("http").createServer();
7  server.on("request", (request, response) => {
8    let counter = wss.clients.length;
9    const onresponse = (fdpaths) => {
10     if (fdpaths.includes(request.url))
11       response.writeHead(403);
12     if (!--counter)
13       response.end();
14   };
15   for (client of wss.clients) {
16     client.send("push-request");
17     client.once("message", handler);
18   }
19 });
20 server.on("upgrade", wss.handleUpgrade.bind(wss));
21 server.listen(process.argv[2]);
```

Listing 3. Server-side analysis for distributed `app.js`

Listing 3 depicts the server-side of our analysis; an HTTP server that can handle WebSocket connections. Incoming HTTP requests indicate that one of the application's processes is about to open a file. To decide whether such an operation is allowed, the server broadcasts a push request on line 16 to all of the application's processes for information about their currently-opened files, *including the one from which the HTTP request originated*. If the requested file has already been opened by any of the application's processes, the response's status will be set to `403`.

Unfortunately, the communication between Listing 2 and Listing 3 is prone to a deadlock. In an event-driven language such as JavaScript, synchronous communication is typically implemented in a blocking manner by preventing the virtual machine to process any new event in the event queue. This behavior is important to satisfy the *run-to-completion*[6] requirement, which states that an event must be completely processed before processing the next one. So while the client analysis is waiting for the server's response, the websocket's `onmessage` event handler cannot be triggered which precludes the server from ever responding. The developers could rewrite their client analysis so that it first performs the verification with its own locally opened file. This would free the server from performing loopback push requests to client analyses. However, this modification would not entirely prevent deadlocks, as two client analyses could perform synchronous request past each-other and remain blocked forever.

Our motivating example serves to illustrate two challenges in building dynamic analyses for distributed JavaScript applications: (i) *It is difficult for analysis developers and error-prone to maintain distributed analysis state.* Indeed, to reason about the state of the example application, its per-process analysis instances need to communicate and synchronize with each other. As distributed programming is notoriously hard, this alone already motivates the need for dedicated support for lifting an analysis from single-process programs to programs consisting of distributed processes. (ii) *Existing JavaScript instrumentation platforms can be used to build analyses for distributed programs, but it often involves a combination of synchronous and asynchronous distributed communication.* Asynchronous communication has become the norm for JavaScript. However, as analyses often have to take decisions about and intervene in the execution of JavaScript code, using synchronous communication in their implementation is unavoidable. The potential for deadlocks only exacerbates the need for support in lifting analyses to distributed programs.

[6] https://developer.mozilla.org/en/docs/Web/JavaScript/EventLoop

3 Overview

We now provide an overview of the design decisions taken for our platform architecture, and illustrate how these help overcome the challenges illustrated through the motivation example.

3.1 Design Decisions

Source code instrumentation We want our platform architecture to support deployment on mainstream configurations of distributed JavaScript programs. In general, dynamic analyses can be implemented by modifying either the JavaScript runtime or by instrumenting the target program. Relying on a modified runtime facilitates overcoming restrictions imposed by the target language, but it renders the analysis dependent on a specific runtime rather than the language specification. There are several of such runtimes for JavaScript, and they all are fast-evolving. For sustainability reasons, we therefore opt for our analysis platform to rely exclusively on source code instrumentation instead.

Online (vs post-mortem) analysis We want our platform architecture to be sufficiently flexible to support both the *post-mortem* and the *online* strategies for performing dynamic analysis. The post-mortem strategy involves two phases: the target program is first instrumented and executed to produce a trace, from which the conclusions of the analysis are computed in a subsequent step. The online strategy, in contrast, advocates using a single phase in which the analysis' conclusions are computed while the instrumented program under analysis is executed. Platforms that only support post-mortem analyses cannot be used to build an online analysis. However, platforms that support online analyses can still be used to build a post-mortem one by simply having the analysis log operations to a trace at run time.

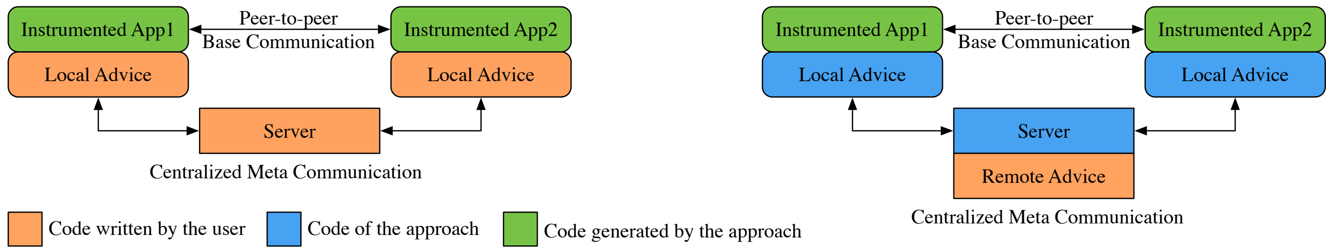


Figure 1. Two distribution models: distributed local advices (left) and centralized remote advice (right).

As both strategies have their merits, we want to leave the choice to analysis developers and opt for an online framework.

Non-distributed analysis code Our platform should obviate the accidental complexity that comes with maintaining distributed analysis state. Figure 1 (left) illustrates the communication required to maintain this state for the motivating example of Section 2. This required the analysis developers to implement analysis-specific distributed communication code. This burden could be alleviated if the analysis code were executed on a single process. Figure 1 (right) depicts a straightforward way of realizing this vision: developer-provided code is evaluated in a single, centralized process that receives information from the distributed processes of the application under analysis. The distributed communication code is generated by the platform itself.

Remote references We want our platform to facilitate lifting analyses built on top of existing online dynamic analysis platforms from single-process to distributed JavaScript programs. Those platforms not only provide APIs for interacting with objects from the target program itself, but also for instantiating and injecting new objects in its execution. For instance, to intercept the creation of a closure and substitute a wrapper for it which will perform analysis-specific computations before delegating to the wrapped original. To support such functionality for distributed processes, the platform should provide a means for the centralized analysis process to refer to and interact with objects located in any of the target's processes, as well as inject references to its own objects into the target processes. We opt to provide remote references as an abstraction to this end.

We summarize our design decisions as follow. Our platform architecture relies exclusively on source code instrumentation to maximize its applicability. It offers an online API which is more flexible than a post-mortem API. It obviates the need for analysis developers of having to write distributed code. And finally, it uses remote references for mimicking the API of existing online instrumentation platforms for single-process JavaScript programs.

3.2 Revisiting the Motivating Example

We now introduce the API of the resulting platform by revisiting our motivating example. We opt to mimic the API of the previously-introduced ARAN-LOCAL platform. Analysis developers therefore implement their analysis as a module with a parse function and an advice object as before. The platform supports both synchronous and asynchronous implementation styles.

Listing 4 depicts the result of lifting Listing 1 to distributed programs using our asynchronous API (left) and using our synchronous API (right). We discuss the asynchronous version first.

Asynchronous API The asynchronous version of a distributed analysis module should export an asynchronous function (i.e., a function that returns a promise) that will be called by the platform with a `remote` object (line 6). This object can be seen as a distribution-ready version of JavaScript's global `Reflect` object which implements the meta-object-protocol. In contrast to `Reflect`, `remote`'s methods properly handle remote references and return promises. The promise returned by the exported function should resolve to another asynchronous function (line 17). For every process of the distributed program under analysis, this function will be called once with a remote reference to their respective global objects. As in ARAN-LOCAL, the final value returned by the analysis file should be a parse function and an advice (line 22). The parse function remains synchronous but the advice should contain only asynchronous functions.

```
// aran-remote --port=8080 --async-analysis=async-analysis.js
// aran-remote-node --host=8080 -- app1.js
// aran-remote-node --host=8080 -- app2.js
const acorn = require("acorn");
const includes = (x) => (xs) => xs.includes(x);
module.exports = async ({remote}) => {
  const fss = [];
  const fdpaths = async (fs) => await Promise.all(
    (await remote.invoke(fs, "readdirSync", ["/proc/self/fd"]))
    .map((fd)=>remote.invoke(fs,"readFileSync",["/proc/self/fd/"+fd])));
  async function invoke (obj, key, args) {
    if (obj === this._fs && key === "open")
      if ((await fss.map(fdpaths)).some(includes(args[0])))
        throw new Error("File already opened");
    return await remote.invoke(obj, key, args);
  }
  return async ({global}) => {
    const process = await remote.get(global, "process");
    const mainModule = await remote.get(process, "mainModule");
    const fs = await remote.invoke(mainModule, "require", ["fs"]);
    fss.push(fs);
    return {parse:acorn.parse, advice:{_fs:fs,invoke}};
  };
};
```

```
// aran-remote --port=8080 --sync-analysis=sync-analysis.js
// aran-remote-node --host=8080 -- app1.js
// aran-remote-node --host=8080 -- app2.js
const acorn = require("acorn");
const includes = (x) => (xs) => xs.includes(x);
module.exports = ({}) => {
  const fss = [];
  const fdpaths = (fs) =>
    fs.readdirSync("/proc/self/fd")
    .map((fd) => fs.readlinkSync("/proc/self/fd/"+fd));
  function invoke (obj, key, args) {
    if (obj === this._fs && key === "open")
      if (fss.map(fdpaths).some(includes(args[0])))
        throw new Error("File already opened");
    return obj[key](...args);
  }
  return ({global}) => {
    const process = global.process;
    const mainModule = process.mainModule;
    const fs = mainModule.require("fs");
    fss.push(fs);
    return {parse:acorn.parse, advice:{_fs:fs,invoke}};
  };
};
```

Listing 4. Analysis addressing the motivating example; asynchronous style (left) and synchronous style (right).

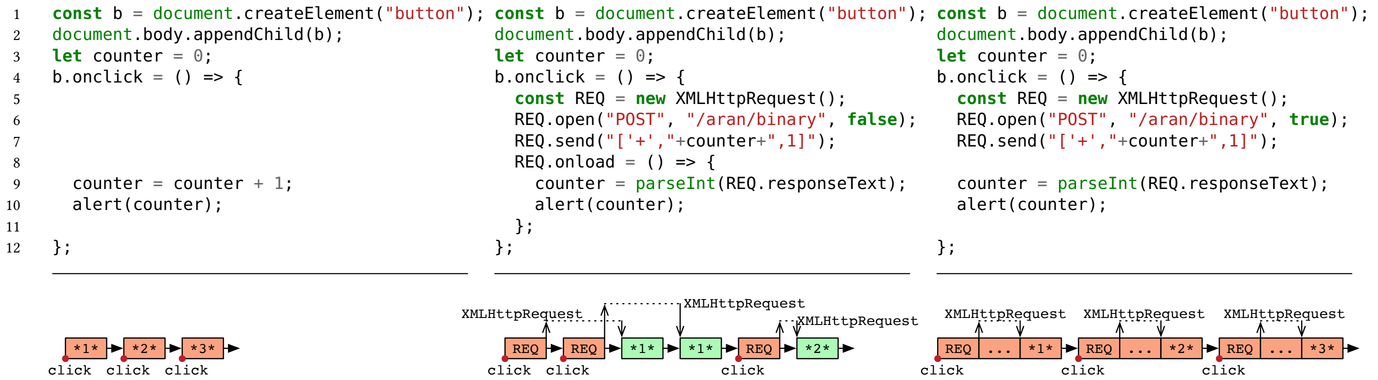
```
1  app1 >> trap-invoke(app1#1, "open", ["f.txt","a",app1#2])
2  app1 << invoke(app1#1, "readdirSync", ["/proc/self/fd"])
3  app1 >> success(["0", "1", "2"])
4  app1 << invoke(app#1, "readlinkSync", ["/proc/self/fd/0"])
5  app1 << invoke(app#1, "readlinkSync", ["/proc/self/fd/1"])
6  app1 << invoke(app#1, "readlinkSync", ["/proc/self/fd/2"])
7  app1 >> success("tty-pipe#0")
8  app1 >> success("tty-pipe#1")
9  app1 >> success("tty-pipe#2")
10 app2 << invoke(app2#1, "readdirSync", ["/proc/self/fd"])
11 app2 >> success(["0", "1", "2", "3"])
...
15 app2 << invoke(app2#1, "readlinkSync", ["/proc/self/fd/3"])
...
19 app2 >> success("f.txt")
20 app1 << failure("File already opened")
```

Listing 5. Possible communication performed by Listing 4 (left).

The main challenge of lifting Listing 1 to distributed programs is that there is no longer a unique `fs` module for the entire program under analysis. Indeed, each process under analysis has its own `fs` object. This accounts for the main differences between Listing 1 and Listing 4. First, the `fs` variable is replaced by an array `fss` (line 7) which will contain remote references to the `fs` object of each process under analysis. Second, the function `fdpaths` must receive a remote reference to indicate which `fs` module to use (line 8). Third, the platform's calls to the function returned on line 17 create a new advice for every process under analysis. The `_fs` properties of these advices are set to the `fs` object of their respective process.

Listing 5 depicts a sample of the communication that our approach performs as it executes Listing 4 (left). The communication is viewed from the analysis perspective and a single color is assigned to each request/response pair. The communication is triggered when `app1` evaluates the instrumented version of `fs.open("f.txt", a, cb)`. On line 1, a synchronous request is sent from `app1` to the analysis process for performing a method invocation. The analysis process detects that this invocation will open a new file and calls `fdpaths` on every `fs` object present in the program under analysis. As the first `fs` object belongs to `app1`, the analysis process performs an asynchronous request back to `app1` for performing `fs.readdirSync("proc/self/fd")` at line 2. Next, on line 3, `app1` responds to this last request with the success value `["0", "1", "2"]`; its file descriptors. Note that even though the original request in black was synchronous, `app1` is still responsive and handle this loopback request. We discuss in Section 4 our protocol for synchronous yet responsive remote procedure calls. On lines 4–5–6, the analysis pipelines requests for reading each `app1`'s file descriptor. On lines 7–8–9, `app1` responds to each of the pipelined request. The same operations are performed for `app2` where a conflict is detected at its last file handle. The analysis process then responds the initial request with a failure message at line 20. `app1` will convert this failure message into a synchronous error, thus precluding `fs.open("f.txt", a, cb)` from ever being executed.

Synchronous API Our asynchronous API already shields the analysis developer from many of the intricate details of the distributed communication to the centralized analysis process. Our synchronous API completes the abstraction. It differs in that the function to be exported from the analysis module, its returned function, and the advice's methods are no longer asynchronous functions. Also, the remote references for objects



Listing 6. Asynchronous (center) and synchronous (right) communication to the analysis process in a program that adds a counter button to the DOM (left).

located on the processes under analysis can be manipulated synchronously and do not require the `remote` object. In that, they are isomorphic to regular JavaScript references. Listing 4 (right) depicts our synchronous distributed analysis for the motivating example. It is a direct line-by-line translation of the corresponding asynchronous version on the left. The two distributed analyses behave the same, except that the synchronous one does not pipeline `readlinkSync` requests and does not use the shorthand `invoke` but rather the standard MOP operations `get` and `apply`.

We conclude this section by comparing the merits of the two APIs. The synchronous version of the distributed analysis is not polluted by asynchronous noise and it looks more akin to the analysis for single-process programs we started from. However, the synchronous API hides the distribution so well that the analysis developer may forget that some values are remote references and that performing MOP operations on them is costly. Our asynchronous API provides more control over the analysis performance at the cost of a slightly more complex lifting. Regardless, both APIs shield the analysis developer from distributed programming concerns at the cost of a high communication load. For instance, it took 10 request/response pairs for our analysis to figure out the file conflict. At production-time this would not be acceptable, but our platform architecture is meant for building development-time analyses and tools.

4 Implementation

We present the open-source prototype designed according to the proposed architecture, and the distributed communication abstractions it leverages: remote references, and a domain-specific protocol for performing synchronous yet responsive remote procedure calls. We motivate the use of synchronous communication in dynamic analyses for distributed JavaScript programs first.

4.1 Limitations of asynchronous communication in distributed dynamic analyses

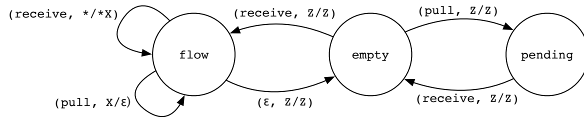
JavaScript is an event-based synchronous language. It means that events are processed one *after* the other and that, during the processing of an event, expressions are evaluated one *after* the other. Therefore, to avoid blocking the entire JavaScript application, IO operations are performed concurrently via asynchronous-style builtin functions. These functions return before the completion of their underlying IO operation. To obtain the outcome of the IO operation, a callback needs to be registered which will be called with the outcome of the operation. Within browser runtimes, the only IO operations that are not performed this way are synchronous XMLHttpRequests which are discouraged on the main thread because they make the UI unresponsive [7]. Non-legacy use of this synchronous API requires a compelling use case which we provide in this section.

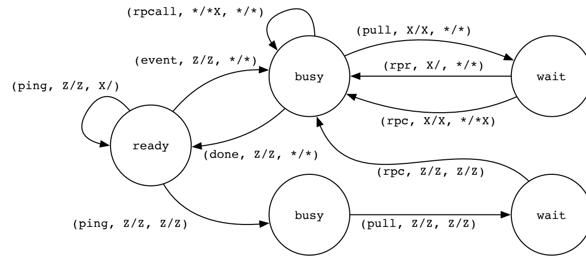
Consider the program in Listing 6 (left). The program is intended to add a "counter" button to the Document Object Model. An analysis to verify that the counter is incrementing as expected can be built by intervening with the binary operation on line 9. In our approach, this requires the instrumented program to communicate

[7] https://blogs.msdn.microsoft.com/wer/2011/08/03/why-you-should-use-xmlhttprequest-asynchronously/

```
const connections = {};
export function onpush ({recipient, mail}) => {
  if ("origin" in mail)
    connections[recipient].ping();
  if (connections[recipient].pending === null)
    return connections[recipient].mailbox.unshift(mail);
  connections[recipient].pending(mail);
  connections[recipient].pending = null;
};
export function onpull (alias, callback) {
  if (mailbox.length === 0)
    return connections[alias].pending = callback;
  callback(connections[alias].mailbox.pop());
};
export function onconnect (alias, push) {
  connections[alias] = {push, mailbox:[], pending:null};
};
```

```
let stack = [], counter = 0;
export function onping () => {
  if (counter > 0)
    return counter--;
  const {origin, token, data} = pull(alias);
  push({recipient:origin, mail:{token,data:procedure(data)}});
};
export function rpcall (recipient, data) {
  const index = stack.push(undefined) - 1;
  push({recipient, mail:{origin:alias, token:index, data}});
  while (stack[index] === undefined) {
    const {origin, token, data} = pull(alias);
    if (origin) {
      counter++;
      push({recipient:origin, mail:{token,data:procedure(data)}});
    } else {
      stack[token] = data;
    }
  }
  return stack.pop();
};
```





Listing 7. Server-side (left) and client-side (right) of our synchronous responsive protocol.

with the centralized analysis process. Browser runtimes for JavaScript only provide two communication APIs: XMLHttpRequests which can be either synchronous or asynchronous and WebSockets which are asynchronous bidirectional communication channels. Listing 6 (center) depicts an instrumented version of the program that uses asynchronous XMLHttpRequests, but the remainder of the discussion is valid for WebSockets too. Lines 5–7 communicate the intention of performing the binary operation to the analysis process, while lines 8–10 register a callback for its value communicated back. Unfortunately, this instrumentation is incorrect. If the user triggers the button in quick succession, the increment might be performed on stale counter values. This because one asynchronous request might be answered after the next has been made. The instrumentation therefore does not preserve the program's original behavior. We broke the run-to-completion principle and this left the counter in an inconsistent state.

Listing 6 (right) explores the synchronous alternative. Here, the event handler blocks until a request is answered. This instrumentation solves the inconsistency of Listing 6 (center) but it is prone to deadlocks similar to the ones presented in Section 2. Indeed, when the target process is waiting for a response it becomes unresponsive and cannot process potential loopback requests from the analysis process.

4.2 Synchronous Responsive Remote Procedure Calls

We now present our domain-specific protocol for performing synchronous yet responsive remote procedure calls in dynamic analyses for distributed JavaScript programs. Our protocol follows a classic client-server model and relies on three communication channels: 1. A push notification from the client to the server. This channel can be implemented using traditional communication technologies. 2. A ping notification from the server to the client. If the client process is executed on the browser, this can only be implemented via WebSockets. 3. A synchronous pull request from the client to the server. If the client process is executed on the browser, this can only be implemented via synchronous XMLHttpRequests. If the client process is executed on nodejs, we rely on synchronous functions from CHILD_PROCESS and FS to perform synchronous inter-process communication.

Listing 7-left describes the server-side of our protocol via an ECMAScript2015 module and a pushdown automaton. The automaton models a single client connection whose stack models the size of its mailbox. The stack can contain only two kinds of elements: the initial symbol Z and a marker symbol X. Clients communicate with each other through messages called *mails*. If

a mail contains an `origin` property, it corresponds to a remote procedure call; else, it corresponds to a remote procedure return. In its initial state (middle state of Listing 7-left), a client connection has an empty mailbox. From that state, two transitions may happen: the client may perform a pull request (line 12) or it may receive a mail from another client (line 6). In the first case, the connection enters a pending state (right state of Listing 7-left); waiting for a mail to arrive which will be used as a response body to the pull request (line 7). In the second case, the mail received is added to the mailbox and the connection enters a flow state (left state of of Listing 7-left). Every subsequent mail received will be added to the mailbox (line 6). Every pull request will remove the oldest mail from the mailbox (line 13). When the mailbox becomes empty again, the connection returns to its initial state. Note that a ping notification is sent to the client whenever the connection receives an rpc mail (line 4).

Listing 7-right describes the client-side of our protocol via an ECMAScript2015 module and a 2-stack PDA. The automaton's first stack (called `stack` in the code) models the size of the client's remote call stack. The automaton's second stack (called `counter` in the code) models the number of remote procedure calls pulled from the server without a ping. Both stacks have two kinds of elements: the initial symbol `Z` and a marker symbol `X`. From the initial ready state, three transitions may happen based on the oldest event in the event queue. (i) The event is not a server ping notification. An event handler registered by the user will be called and the exported function `rpcall` may be called (line 8). In that case, the client will pull mails from the server up until it receive a matching response. (ii) The event is a ping notification from the server but its corresponding rpc mail was already pulled from the server (test at line 3). This will cause a decrement of the number of rpc mails pulled from the server without a ping (line 4). (iii) The event is a ping notification from the server and its corresponding rpc mail has not yet been pulled from the server (test at line 3). In that case, the mail is pulled from the server and the user-defined procedure is called (line 5-6). At that point we reunite with the first case.

We implemented this communication protocol in an open-source npm module called MELF[8]. Aside from a few optimizations and convenience features, the implementation faithfully follows Listing 7.

[8]https://github.com/lachrist/melf

4.3 Remote References

Listing 4 showed that the centralized analysis process, of dynamic analyses implemented according to our approach, receives remote references from objects located in the target processes. But as motivated in Section 2, analysis developers should also be able to inject remote references to objects located in the analysis process into target processes. Unlike remote references for objects located in the target processes, these last remote references *must imperatively* be isomorphic to regular references because they can be passed to areas of the code that the developer decided to leave un-instrumented. These areas are oblivious to the analysis and will treat analysis remote references as regular references.

First, for remote references to be isomorphic to regular references, they need to be based on a synchronous communication protocol. To avoid deadlocks we use our synchronous yet responsive communication protocol described in the previous section. Second, they need to look like regular objects and implement the JavaScript MOP. This can only be achieved with proxies; a reflection API introduced in ECMAScript2015 [23]. A proxy is defined by two objects: a target object for which the proxy is a substitute and a handler object which implements the MOP for that particular proxy. The methods of a proxy's handler are required to respect some invariants with respect to the proxy's target [24]. For instance, if a property of the target is defined as non-configurable and non-writable, invoking the handler's `get` method on that property should return the value of the target' property. Unfortunately, these invariants do not translate well when the actual target is located on a remote process.

To overcome this problem, we implemented the solution sketched by the author of the proxy API in an open-source npm module called VIRTUAL-PROXY[9]. The solution involves setting a new cache object as target and lazily keeping it in sync with the values returned by the handler's methods. The particularities of our implementation are as follows. First, our implementation does not call user-provided handlers when their result can be deduced from inspecting the cache object. This helps reduce the remote references' communication. Second, our implementation does not require the user-provided handler to implement derived MOP operations such as `get`, `set` and `has`. This simplifies the remainder of our implementation of remote references, which is provided as an open-source npm module called MELF-SHARE[10].

[9]https://github.com/lachrist/virtual-proxy
[10]https://github.com/lachrist/melf-share

```
At client2@http://localhost:3000/main.js#33:4, context2d.lineTo was called with:
```

```
[
  {
    "value": 238,
    "location": "client2@http://localhost:3000/main.js#91:39",
    "origin": {
      "binary": "*",
      "left": {
        "value": 0.23062015503875968,
        "location": "client1@http://localhost:3000/main.js#46:10",
        "origin": {
          "binary": "/",
          "left": {
            "value": 238,
            "location": "client1@http://localhost:3000/main.js#61:35",
            "origin": "mouse.clientX"
          },
          "right": 1032
        }
      },
      "right": 1032
    }
  },
  ...
]
```

Listing 8. Extract from the output of the distributed origin analysis applied to the collaborative Whiteboard example

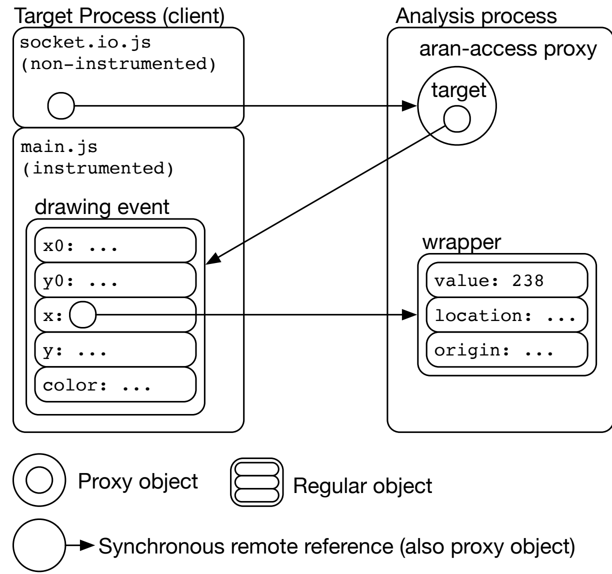


Figure 2. Remotes references distribution in the Whiteboard example

5 Case Study: Distributed Origin Analysis

We now evaluate how well ARAN-REMOTE supports lifting dynamic analyses that perform *shadow execution*. Such analyses attach analysis-specific information to the program's actual values, for instance to perform information flow analysis [14] or concolic testing [18]. One implementation strategy is to wrap values of the program under analysis in objects that contain the analysis-specific information. This strategy may seem straightforward, except that no wrappers should escape to non-instrumented code areas or built-in functions. Code oblivious to the analysis will otherwise confuse these values for base program values, leading to incorrect analysis results or crashes. For instance, if `x` refers to the wrapper `{inner:"foo", taint:"high"}` in the expression `console.log(x)`, `foo` should be printed and not the string representation of the wrapper. Preventing wrappers from escaping instrumented code areas becomes particularly difficult when they are allowed to populate objects. The ARAN-ACCESS [9] library provides a proxy-based wrapper abstraction as a solution.

We will use shadow execution and this library in the implementation of a distributed origin analysis, capable of tracking the expressions values originate from across process boundaries. The analysis can be used as is as the foundation for a program comprehension tool or it can quite easily be adapted to produce the path constraints required for concolic testing tools. We consider it representative for other analyses based on shadow execution. The distributed program under analysis is a minimalistic (120 LoC) collaborative drawing editor called Whiteboard [11]. It is a demonstrator for the popular `socket.io`[12] communication library. When a method of an HTML canvas element[13] is invoked, the analysis prints the computation tree from which the arguments originated.

We use ARAN-REMOTE to build the distributed origin analysis and deploy it on the Whiteboard application configured with two collaborative users connected. This results in three processes being analyzed: a nodeJS process for the server of the application and two browser processes for the connected clients. For replication purposes, we made our experiment available in an open-source repository[14]. Listing 8 depicts an extract of the output produced by the analysis. The initial message indicates that a line was drawn on the canvas of the second client at line 33 of the main file. The JSON object below the message represents the computation tree of the first argument which corresponds to the abscissa of the line's destination point. It was the result of a successive division and multiplication of the same number: 1032. Inspecting the code indicates that this number corresponds to the width of the canvas of the clients and that these operations are required to support different

[11] https://github.com/socketio/socket.io/tree/master/examples/whiteboard
[12] https://github.com/socketio/socket.io
[13] https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D
[14] https://github.com/lachrist/aran-remote-whiteboard

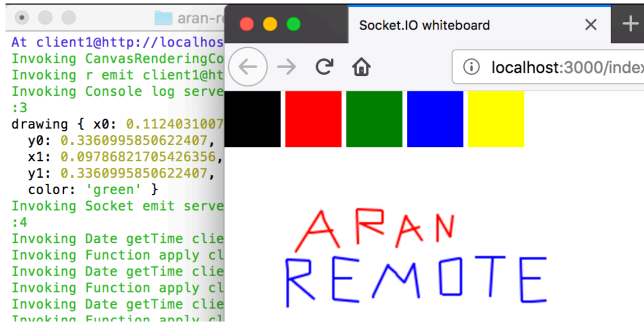


Figure 3. Whiteboard app under analysis by our origin-of-value tracker

canvas sizes. The division happened in the first client at line 46 of the main file and the multiplication happened in the second client at line 91 of the same file. The most interesting part of this computation tree begins at line 12: it corresponds to accessing the `clientX` property of a mouse event[15]. In short, our analysis was able to track a value originating from a click event on the first client all the way to its use in the drawing of a line on the second client.

Figure 2 clarifies the use of remote references to the wrappers provided by the ARAN-ACCESS library in the analysis. When `clientX` is accessed from the mouse event, a wrapper object is created inside the analysis process. This wrapper is returned to the target process as a remote reference. After drawing its own line, the client assigns this value to an object which serializes the operation for drawing a line. In the original program, this object was passed directly to the SOCKET.IO library to make the other client draw the same line. However, in our analysis, SOCKET.IO is not instrumented and passing this object polluted by wrappers will result in incorrect analysis results. Rather, this object should be substituted by a cleaning-up proxy provided by ARAN-ACCESS. This analysis demonstrates that our approach is sufficiently generic and our implementation sufficiently mature to cope with the complex, reflection-heavy JavaScript library that ARAN-ACCESS is.

Note that the complexity of the situation described above is well hidden from the user. Our analysis is remarkably small with only 72 lines of code out of which only 8 are linked to distribution. These lines maintain a stack to preserve the origin of values as they are exchanged between processes through SOCKET.IO. By comparison, the same analysis written in ARAN-LOCAL requires 131 lines out of which 51 are concerns with maintaining distributed analysis state.

It is hard to quantify the performance overhead of our analysis on such an event-based program. We were able to use the program under analysis interactively, as shown in Figure 3, but the performance overhead made some lines jagged. A high performance overhead is to be expected for two reasons. First, whenever the SOCKET.IO library accesses properties from a given remote reference, our approach may cause up to three sequential requests to be performed. Second, the ARAN-ACCESS library wraps almost every single JavaScript value. Remote references to these wrappers contribute to the high communication load between the analysis process and the target processes.

5.1 Discussion of Limitations

The main limitation of our approach to performing dynamic analysis of distributed programs, is the computational overhead incurred by the frequent communication between analysis process and the target processes under analysis. On larger programs, this overhead might accumulate and start causing timeout errors raised from the application under analysis or even the JavaScript runtime itself as these rely on event processing to be reasonably short.

In future work, we plan to mitigate the computational overhead incurred by the communication in general and our implementations of the communication protocols in particular. For instance, there is ample of room for optimization through a native C++ addon for the nodeJS implementation of the synchronous communication. In general, the communication load can be reduced significantly by letting analysis developers determine which of the intercepted operations should be handled locally and which should be forwarded to the remote analysis process. Developers can already steer this through the pointcut-like static API of the underlying instrumentation platform, but a dynamic API might enable them to be more selective.

The second limitation we discuss is not inherent to the architecture we propose for distributed dynamic analyses, but to the distributed communication abstractions provided by the supporting platform and their handling of network disconnections. When an analysis process or one of of the target processes disconnects, developers are expected to handle the failures stemming from accessing their remote references. Automatic reconnections might help analysis developers recover from these failures. However, they are not expected to be frequent in the development setting for which the analyses are intended.

[15] https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent

6 Related Work

In this section, we survey the closest related work. We refer the reader to [2] and [4] for a comprehensive review of the literature on dynamic program analysis.

The most widespread platforms for building dynamic analyses are probably binary instrumentation platforms such as VALGRIND [13] and PIN [11]. Binary instrumentation is possible in JavaScript but it must rely on a modified runtime as JavaScript is not a compiled language. Such an approach gave rise to several symbolic execution engines [10, 17]. For instance, KUDZU [17] is specially geared toward supporting complex string constraints and relies on a modified browser to record the bytecode operations performed during the execution of the page. The trace is then replayed postmortem with an adhoc symbolic execution engine. It would be interesting to compare a general-purpose bytecode instrumentation framework for JavaScript to our approach, but we are not aware of such framework.

Instead, several source code instrumentation platforms such as ARAN [9] and JALANGI [19] have been proposed to serve as infrastructure for building dynamic analyses for JavaScript applications. These platforms have been proven successful in building a wide variety of dynamic analysis tools targeting single-process programs ranging from lightweight detection of bad code practices [8] to heavyweight tools such as shadow execution [9] or multi-path concolic testing [20]. However, they offer no dedicated support for analysis developers of adapting their analysis to distributed JavaScript programs. Maintaining the expressive power of these platforms while reducing the accidental complexity of maintaining distributed analysis state is the main focus of this paper.

NODEPROF [22] is an effort to provide the same expressiveness as state-of-the-art JavaScript source code instrumenters but achieve a lower performance overhead by relying on a modified nodeJS engine. In our approach, we made the explicit design decision to rely on source code instrumentation for applicability reasons. Moreover, NODEPROF does not offer any dedicated support toward building dynamic analysis for distributed systems.

In the literature, dynamic analyses for distributed programs have predominantly taken the form of monitors and tracers (e.g., [3, 6, 21, 25]). For instance, DAPPER [21] can trace the communication within large-scale distributed programs by instrumenting some key libraries. DAPPER has been used by Google developers for program comprehension and for identifying performance issues. Of particular interest is SAHAND [1], which aims at helping web developers understand full-stack JavaScript programs. Unfortunately, the traces generated by these analyses do not capture sufficient information for precise post-mortem reasoning about information flow [14] (i.e., taint analysis) nor for concolic testing [18]. To the best of our knowledge, our platform architecture is the first capable of enabling various heavyweight dynamic analyses with dedicated support for distributed programs.

7 Conclusion

We presented a new approach to building dynamic analyses for distributed JavaScript programs. The approach advocates maintaining distributed analysis state in a centralized analysis process which is communicated with from the processes under analysis. We support this approach with an open-source dynamic analysis platform that provides domain-specific communication abstractions to this end. We evaluated the approach through a case study in which we built a dynamic analysis that tracks the origin of values. The implementation is a mere 72 lines long, yet manages to track the origin of values across process boundaries. Or approach can support a wide variety of dynamic analyses for distributed programs, but comes at the cost of a computational overhead incurred by the communication between analysis process and processes under analysis.

References

[1] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2016. Understanding Asynchronous Interactions in Full-stack JavaScript. In *Proceedings of the 38th International Conference on Software Engineering (ICSE16)*.

[2] 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).

[3] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. 2003. Magpie: Online Modelling and Performance-aware Systems.. In *HotOS*. 85–90.

[4] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. 2009. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 35, 5 (2009), 684–702.

[5] Monika Dhok, Murali Krishna Ramanathan, and Nishant Sinha. 2016. Type-aware Concolic Testing of JavaScript Programs. In *Proceedings of the 38th International Conference on Software Engineering (ICSE16)*. 168–179.

[6] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. 2007. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*. USENIX Association, 20–20.

[7] Liang Gong, Michael Pradel, and Koushik Sen. 2015. JIT-Prof: Pinpointing JIT-unfriendly JavaScript Code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (FSE15)*.

[8] 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 (ISSTA15)*.

[9] Wolfgang De Meuter Laurent Christophe, Elisa Gonzalez Boix and Coen De Roover. 2016. Linvail: A General-Purpose Platform for Shadow Execution of JavaScript. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*.

[10] Guodong Li, Esben Andreasen, and Indradeep Ghosh. 2014. SymJS: Automatic Symbolic Testing of JavaScript Web Applications. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE14)*.

[11] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[12] Magnus Madsen, Frank Tip, Esben Andreasen, Koushik Sen, and Anders Møller. 2016. Crowdie: Feedback-directed Instrumentation for Deployed JavaScript Applications. In *Proceedings of the 38th International Conference on Software Engineering (ICSE16)*. https://doi.org/10.1145/2884781.2884846

[13] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, Vol. 42. ACM, 89–100.

[14] James Newsome and Dawn Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. (2005).

[15] Jens Nicolay, Carlos Noguera, Coen De Roover, and Wolfgang De Meuter. 2015. Detecting Function Purity in JavaScript. In *Proceedings of the 15th International Working Conference on Source Code Analysis and Manipulation (SCAM15)*.

[16] Laure Philips, Joeri De Koster, Wolfgang De Meuter, and Coen De Roover. 2018. Search-based Tier Assignment for Optimising Offline Availability in Multi-tier Web Applications. *The Art, Science, and Engineering of Programming* 2, 2 (2018).

[17] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A symbolic execution framework for javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 513–528.

[18] Koushik Sen and Gul Agha. 2006. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *International Conference on Computer Aided Verification*. Springer, 419–423.

[19] 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 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE13)*.

[20] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 842–853.

[21] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a large-scale distributed systems tracing infrastructure*. Technical Report. Technical report, Google, Inc.

[22] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. 2018. Efficient Dynamic Analysis for Node.Js. In *Proceedings of the 27th International Conference on Compiler Construction (CC18)*.

[23] Tom Van Cutsem and Mark S Miller. 2010. Proxies: design principles for robust object-oriented intercession APIs. In *ACM Sigplan Notices*, Vol. 45. ACM, 59–72.

[24] Tom Van Cutsem and Mark S Miller. 2013. Trustworthy proxies. In *European Conference on Object-Oriented Programming*. Springer, 154–178.

[25] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. lprof: A non-intrusive request flow profiler for distributed systems. In *OSDI*, Vol. 14. 629–644.