2018

# Righting Web Development

John Vilk

# RIGHTING WEB DEVELOPMENT

A Dissertation Presented

by

JOHN VILK

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2018

College of Information and Computer Sciences

# RIGHTING WEB DEVELOPMENT

A Dissertation Presented

by

JOHN VILK

Approved as to style and content by:

Emery D. Berger, Chair

Yuriy Brun, Member

Arjun Guha, Member

David Irwin, Member

James Mickens, Member

James Allan, Chair of Faculty
College of Information and Computer Sciences

# DEDICATION

*For my mother, who has always believed in me and pushed me to do my best.*

# ACKNOWLEDGMENTS

First, I would like to thank my advisor, Emery Berger, for his unwavering support and invaluable guidance over the past seven years. Your advice has had a considerable impact on how I conduct research, present ideas, and measure success. You taught me how to test and have confidence in my ideas, and to doggedly pursue them when it is apparent that they are valuable. I consider myself exceedingly lucky that I had the opportunity to work with you. Thank you.

In addition, I am grateful for the support and camaraderie of many students in the PLASMA lab, including Charlie Curtsinger, Dan Barowy, Bobby Powers, Emma Tosch, Sam Baxter, Rian Shambaugh, Breanna Devore-McDonald, Tongping Liu, Rachit Nigam, Tony Ohmann, and Ted Smith. From movie nights to deadline nights, you made life in the lab fun and interesting. Thank you for enduring my endless puns.

The fantastic CICS administration sheltered me from the administrative realities of a state university, and I thank them for that. Leeanne Leclerc ensured that I signed up for healthcare and the credits I needed to graduate every semester, and could be relied upon to know the answer to any administrative question. Laurie Downey made it easy to seek reimbursement for travel and to purchase equipment. The CSCF staff quickly handled requests for projectors, posters, and maintenance, and provided many valuable services that I definitely took for granted.

I would also like to thank my research mentors and colleagues who have shaped me as a researcher, writer, public speaker, and human being. In particular, I would like to thank James Mickens, Ben Livshits, Arjun Guha, Yuriy Brun, Scott Kaplan, David Molnar, and Stephen Freund.

I do not know where I would be today if I did not have the loving support and guidance of my parents, Mary-ann and Chris. Throughout my childhood, they pushed me to work hard and to seek opportunities through higher education. They set me on a path that

led me, a public schooled first generation college student, to complete a PhD in Computer Science.

Finally, I would like to thank my wife Naomi for her support, and for periodically taking me on refreshing vacations that took my mind off of work. Thank you for tolerating my anxiety-riddled behavior on stressful deadlines, and for making me laugh on a regular basis.

# ABSTRACT

## RIGHTING WEB DEVELOPMENT

SEPTEMBER 2018

JOHN VILK

B.Sc., WORCESTER POLYTECHNIC INSTITUTE

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Emery D. Berger

The web browser is the most important application runtime today, encompassing all types of applications on practically every Internet-connected device. Browsers power complete office suites, media players, games, and augmented and virtual reality experiences, and they integrate with cameras, microphones, GPSes, and other sensors available on computing devices. Many apparently native mobile and desktop applications are secretly hybrid apps that contain a mix of native and browser code. History has shown that when new devices, sensors, and experiences appear on the market, the browser will evolve to support them.

Despite the browser's importance, developing web applications is exceedingly difficult. Web browsers organically evolved from a document viewer into a ubiquitous program runtime. The browser's scripting language for web designers, JavaScript, has grown into the only universally supported programming language in the browser. Unfortunately, JavaScript is notoriously difficult to write and debug. The browser's high-level and event-driven I/O interfaces make it easy to add simple interactions to webpages, but these same interfaces lead to nondeterministic bugs and performance issues in larger applications. These bugs are challenging for developers to reason about and fix.

This dissertation revisits web development and provides developers with a complete set of development tools with full support for the browser environment. McFly is the first time-traveling debugger for the browser, and lets developers debug web applications and their visual state during time-travel; components of this work shipped in Microsoft's ChakraCore JavaScript engine. BLeak is the first system for automatically debugging memory leaks in web applications, and provides developers with a ranked list of memory leaks along with the source code responsible for them. BCause constructs a causal graph of a web application's events, which helps developers understand their code's behavior. Doppio lets developers run code written in conventional languages in the browser, and Browsix brings Unix into the browser to enable unmodified programs expecting a Unix-like environment to run directly in the browser. Together, these five systems form a solid foundation for web development.

## TABLE OF CONTENTS

**Page**

**CHAPTER**

**APPENDICES**

# LIST OF TABLES

# LIST OF FIGURES

**Figure** **Page**

# INTRODUCTION

A web browser is present on nearly every internet-connected device, making the browser one of the most important application platforms today. The browser lets developers write code once and run it nearly literally anywhere. Even many apparently native mobile and desktop applications are secretly hybrid apps that contain a mix of native and browser code.

Despite its popularity, the browser presents a challenging environment for writing, debugging, and optimizing web applications. Web applications must be written in JavaScript, which is completely event-driven, designed for concurrency, and extraordinarily dynamic. As a result, web applications can have complicated behavior that is difficult to debug, optimize, and understand. The browser's complex high-level runtime environment coupled with JavaScript's dynamism makes it easy for developers to accidentally introduce memory leaks while simultaneously obscuring their root cause. Developers cannot re-use or easily port code written in traditional programming languages, such as Java and C++, because the browser exposes unfamiliar and incompatible abstractions for a variety of common tasks. Existing browser development tools provide little assistance for these issues because they use techniques designed for conventional runtime environments.

To address these issues, this dissertation presents a complete set of tools that form a solid foundation for web development. These tools make it easier for developers to write, debug, optimize, and understand their web applications.

### McFly: Time-Travel Debugging for the Web

Time-traveling debuggers offer the promise of simplifying debugging by letting developers freely step forwards *and backwards* through a program's execution. However, web applications present multiple challenges that make time-travel debugging especially difficult. A time-traveling debugger for web applications must accurately reproduce all network interactions, asynchronous events, and visual states observed during the original execution, both while stepping forwards and backwards at interactive speeds. This dissertation presents McFly, the first time-traveling debugger for web applications. McFly departs from previous approaches by operating on a high-level representation of the browser's internal state. This approach lets McFly provide accurate time-travel debugging—maintaining JavaScript and visual state in sync at all times–at interactive speeds. I have implemented McFly as an extension to Microsoft Edge, and core parts of McFly have been integrated into a time-traveling debugger for Microsoft's ChakraCore JavaScript engine [89].

### BLeak: Automatically Debugging Memory Leaks in Web Applications

Web application memory leaks can take many forms, including failing to dispose of unneeded event listeners, repeatedly injecting iframes and CSS files, and failing to call cleanup routines in third-party libraries. Leaks degrade responsiveness and can even lead to browser tab crashes by exhausting available memory. Because previous leak detection approaches designed for conventional C, C++ or Java applications are ineffective in the browser environment, tracking down leaks currently requires intensive manual effort by web developers. This dissertation introduces BLeak, the first system for automatically debugging memory leaks in web applications [144]. BLeak's algorithms leverage the observation that in modern web applications, users often repeatedly return to the same (approximate) visual state (e.g., the inbox view in Gmail). Sustained growth between round trips is a strong indicator of a memory leak. To use BLeak, a developer writes a short script to drive a web application in round trips to the same visual state. BLeak then automatically generates a list of leaks found along with their root causes, ranked by return on investment. Guided by BLeak, I identify and fix over 50 memory leaks in popular libraries and apps including Airbnb, AngularJS, Google Analytics, Google Maps SDK, and jQuery.

### BCause: Causal Program Understanding for Web Applications

It is challenging for developers and automated tools to reason about control flow in web applications because the browser is notoriously asynchronous. In particular, web application execution is completely event-driven, and it is often challenging to reason about why events occur in the first place. This dissertation introduces BCause, a framework for understanding the causality of JavaScript events in web applications. During a browsing session, BCause constructs a trace of events and web application actions that cause events. BCause uses the trace to construct a causal graph of the JavaScript events that occur in an execution, letting developers and automated tools trace events back to their root causes. Using BCause, I build a tool called AdBlame that uses causal graphs and ad blocker filter lists to accurately predict the bandwidth consumption of a web application with an ad blocker enabled.

### Doppio: Breaking the Browser Language Barrier

Web applications must be written in JavaScript, the native language of the browser. Existing code written in traditional programming languages cannot be directly translated to JavaScript because the browser is missing traditional operating system abstractions required for that code to run. This dissertation presents Doppio, a runtime system written in JavaScript that breaks the browser language barrier [143]. Doppio emulates single-process POSIX abstractions including threads, an extensible file system, outgoing TCP sockets, and unmanaged memory on top of the unfamiliar resources that the browser already provides. I demonstrate Doppio's usefulness with two case studies: I extend Emscripten with Doppio, letting it run an unmodified C++ application in the browser with full functionality, and present DoppioJVM, an interpreter that runs unmodified JVM programs directly in the browser.

### Browsix: Bridging the Gap Between Unix and the Browser

Although Doppio lets additional programming languages run on the web with full functionality, it does not act as an operating system and cannot be used to port multi-process systems or servers into the browser. This dissertation presents Browsix, a framework that bridges the considerable gap between conventional operating systems and the browser, enabling unmodified programs expecting a Unix-like environment to run directly in the browser [111]. Browsix comprises two core parts: (1) a JavaScript-only system that makes core Unix features (including pipes, concurrent processes, signals, sockets, and a shared file system) available to web applications; and (2) extended JavaScript runtimes for C, C++, Go, and Node.js that support running programs written in these languages as processes in the browser. I illustrate Browsix's capabilities via case studies that demonstrate how it eases porting legacy applications to the browser and enables new functionality.

# CHAPTER 1
# BACKGROUND

Over a period of decades, the browser organically evolved from a static document viewer into an application runtime with the complexity of an operating system. As shown in Figure 1.1, a web application's state is distributed among components including a JavaScript engine that controls computation and a layout engine that controls the user interface and other forms of I/O. This chapter briefly documents the browser's evolution in order to concretely define web applications, provides an overview of the browser's architecture and core features, and describes modern web development tools.

## 1.1 From Web Pages to Applications

While many still think of web pages in terms of an HTML document located at a specific URL, many web pages today like YouTube, Weather.com, and Gmail are web applications that generate their UI inside the browser using JavaScript. It is the latter type of web page that this dissertation is concerned with.

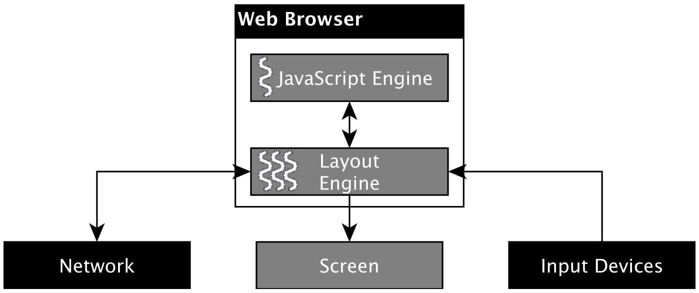


Figure 1.1: A system diagram of a modern web browser. All major web browsers separate web application computation (JavaScript engine) from I/O (Layout Engine). The layout engine is multithreaded, while JavaScript code executes on a single hardware thread.

### 1.1.1 Web Pages

When Tim Berners-Lee wrote the first web browser in 1990, web sites were composed of static hypertext (HTML) documents, or web *pages*, stored on an HTTP server's file system. Beyond clicking hyperlinks to navigate between documents, these web pages were not interactive, and served only as a vehicle for conveying information to their readers.

A few years later, the common gateway interface (CGI) made it possible to write programs, known as CGI scripts, that run server-side and generate web pages on the fly [114]. CGI scripts could also process data that users submit via HTML forms, which let developers add basic interactivity to web sites. However, individual web pages were still completely static once loaded. Although a user could type text into a form field, the browser itself processed the key presses and rendered the resulting text. When the user submits a form, the browser sends data to the server and completely reloads the web page, letting the CGI script running server-side process the form data and re-generate the web page in response. Many staples of the early web were built using CGI scripts, including message boards, guestbooks, visitor counters, and online stores [119].

### 1.1.2 Web Applications

With the introduction of JavaScript in the mid-90's, web pages themselves became interactive, leading to a shift from web pages to web *applications*. JavaScript lets a web page perform arbitrary computations, respond to user interactions, and change its presentation at-will without contacting the server or reloading the web page. While the precise definition of a web application is not universally agreed upon, this dissertation defines a web application to be a web page that uses JavaScript to control its presentation and interactions.

Web applications are not only used on websites; many native and mobile applications are also written completely or partially using HTML, CSS, and JavaScript, and run a web browser internally. By writing an application in this manner, developers can target multiple platforms and devices using the same codebase. Beyond the packaging step required to bundle the application into an executable and a few platform-specific APIs, the process of developing these applications is identical to that for a traditional web application and faces many of the same challenges discussed in this dissertation.

## 1.2 Browser Architecture and Core Features

The browser provides developers with a feature-rich environment that enables a wide variety of applications. However, browser engineers retrofit this environment into an architecture that was originally designed for viewing documents. This section describes this architecture, which is also shown in Figure 1.1, along with its defining features and limitations.

### 1.2.1 Browser Overview

Every major web browser contains a JavaScript engine that performs pure JavaScript computation (§1.2.2) and a layout engine that interacts with the OS to provide functionality such as network requests, timers, and the GUI (§1.2.3): V8 [50] and Blink [27] (for Chrome), JavaScriptCore [8] and WebKit [10] (for Safari), SpiderMonkey [99] and Gecko [93] (for Firefox), and Chakra [88] and EdgeHTML [107] (for Edge). The layout engine exposes this functionality to JavaScript code through a set of JavaScript interfaces, commonly referred to as the Document Object Model (DOM) [56]. The DOM provides JavaScript code with the ability to perform various I/O operations (§1.2.4), many of which are asynchronous, leading to pervasive event-based concurrency (§1.2.5). The layout engine does not provide JavaScript code with access to shared memory multithreading. Instead, the layout engine supports shared-nothing worker threads to enable parallel processing, and proposed extensions to the web platform add support for a restricted form of shared memory parallelism (§1.2.6).

### 1.2.2 JavaScript Engine

JavaScript code executes within the JavaScript engine, which processes events from a single-threaded event loop that is controlled by the browser's layout engine (shown in Figure 1.1). The layout engine produces JavaScript events in response to external events, such as timers firing. To subscribe to an event, JavaScript code uses a DOM interface to provide the layout engine with a function as an event handler. The layout engine places each JavaScript event into the *event queue* where it waits to be processed. The JavaScript thread processes these events and invokes event handlers one at a time.

While JavaScript code executes, the layout engine locks the UI to prevent concurrent user input, such as typing into a form field. As a corollary, long-running events degrade the responsiveness of the webpage. If an event takes too long to execute, the web browser provides the user with the option of killing the unresponsive web application.

### 1.2.3 Layout Engine

The layout engine is the heart of the browser, and is responsible for a web application's input and output. The layout engine provides JavaScript code with the DOM, which contains high-level interfaces for timers, network requests, GUI manipulation, and more. The layout engine also processes HTML, CSS, and user input, which impact DOM-visible state independent from JavaScript execution. DOM interfaces reflect and manipulate state that the layout engine manages and can update independent of JavaScript execution.

For example, the DOM represents the GUI as a tree of HTML elements. JavaScript code accesses elements in the tree as JavaScript objects via the `document` object, but these objects are purely reflective proxies into layout engine state. Reading the `offsetLeft` property on a GUI element triggers a query to the layout engine, which calculates how far left the element is relative to a parent element.

### 1.2.4 I/O Abstractions

Unlike conventional program environments, browsers do not provide web applications with access to traditional operating system resources. Instead, browsers expose a variety of abstractions for tasks that have organically grown out of the needs of the web. For example, instead of a file system, browsers include storage interfaces that provide key-value and object database abstractions. Instead of raw sockets, browsers include an interface that makes HTTP requests, and an interface that establishes an outgoing full-duplex message-based WebSocket connection to a server over TCP.

### 1.2.5 Event-based Concurrency

While JavaScript is single-threaded, Figure 1.1 shows that the layout engine is not. The layout engine exposes asynchronous APIs in JavaScript that dispatch tasks that run concurrently with JavaScript execution, such as network requests, timers, and persistent storage requests. These tasks communicate with the application via events, which the layout engine enqueues in parallel with JavaScript execution. Thus, web applications are actually highly concurrent; given the same application input, events can execute in different orders across executions.

### 1.2.6 Worker Threads

In modern browsers, JavaScript code can spawn worker threads that execute within their own isolated JavaScript context. These worker threads do not share any memory with the main thread or each other, and can only communicate with their parent through a bidirectional and asynchronous message passing interface. Worker threads also have no access to the application's GUI and have their own separate event queues. Counterintuitively, worker threads are a resource provided by the layout engine, which relays messages between JavaScript contexts.

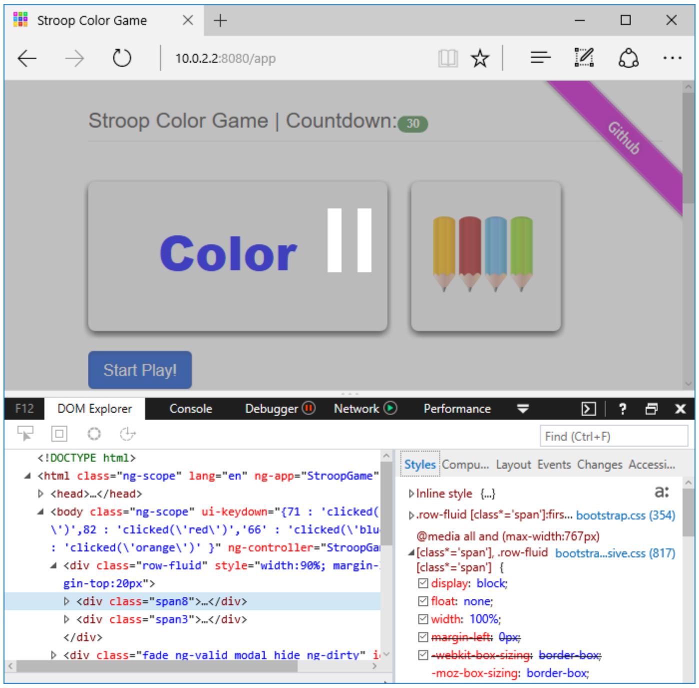
Upcoming changes to the browser enable a restricted form of shared memory between JavaScript contexts. With these changes, multiple JavaScript contexts can share a `SharedArrayBuffer`, which encapsulates a binary buffer of memory. Like the existing `ArrayBuffer` interface, these buffers can be interpreted as an array of specific numeric types, such as 8-bit unsigned integers and 64-bit floating point numbers. In addition, browsers will provide atomic operations on the contents of `SharedArrayBuffer`, including futex primitives [26].

## 1.3 Web Development Tools

Web developers use a variety of tools to write, debug, and optimize their applications. Some of these tools are integrated with the browser itself to provide deeper support for visual state that the layout engine manages. These tools are largely adaptations of conventional software development tools.

### 1.3.1 Debugging Correctness Issues

When a web application is not behaving in the way the developer intended, web developers have only a few basic tools at their disposal to help them track down bugs.



```
<!DOCTYPE html>
<html class="ng-scope" lang="en" ng-app="StroopGame"
  <head>…</head>
  <body class="ng-scope" ui-keydown="{71 : 'clicked(
  \')',82 : 'clicked(\'red\')','66' : 'clicked(\'blu
  : 'clicked(\'orange\')' }" ng-controller="StroopGa
    <div class="row-fluid" style="width:90%; margin-
    gin-top:20px">
      <div class="span8">…</div>
      <div class="span3">…</div>
    </div>
    <div class="fade ng-valid modal hide ng-dirty" i
```

Inline style {…}

.row-fluid [class*='span']:firs... bootstrap.css (354)

@media all and (max-width:767px)
[class*='span'], .row-fluid [class*='span'] { bootstra...sive.css (817)
☑ display: block;
☑ float: none;
☑ width: 100%;
☑ ~~margin-left: 0px;~~
☑ ~~-webkit-box-sizing: border-box;~~
-moz-box-sizing: border-box;

Figure 1.2: A web application running in Microsoft Edge and paused on a breakpoint. Standard browser debuggers let developers inspect DOM state and view the application's GUI at breakpoints.

**Console printing**: One classic debugging technique is to add print statements to a program that log potentially informative messages to standard output; this practice is commonly called `printf` debugging. Developers can also use this technique in the browser. While web applications lack access to standard output streams, browsers contain a hidden web console that web applications can print to using the `console` JavaScript interface. Developers can view the console using the browser's development tools.

**Stepping debugger**: All modern web browsers contain a conventional stepping debugger that let developers set breakpoints, single-step through JavaScript execution, and inspect live application state [9, 16, 90, 94]. The debugger displays application state stored in the JavaScript engine and the layout engine, and provides a JavaScript REPL for more advanced queries. Figure 1.2 displays a screenshot of Microsoft Edge's debugger, which lets the developer examine the DOM while the application is paused on a breakpoint. This feature is standard across browser debuggers.

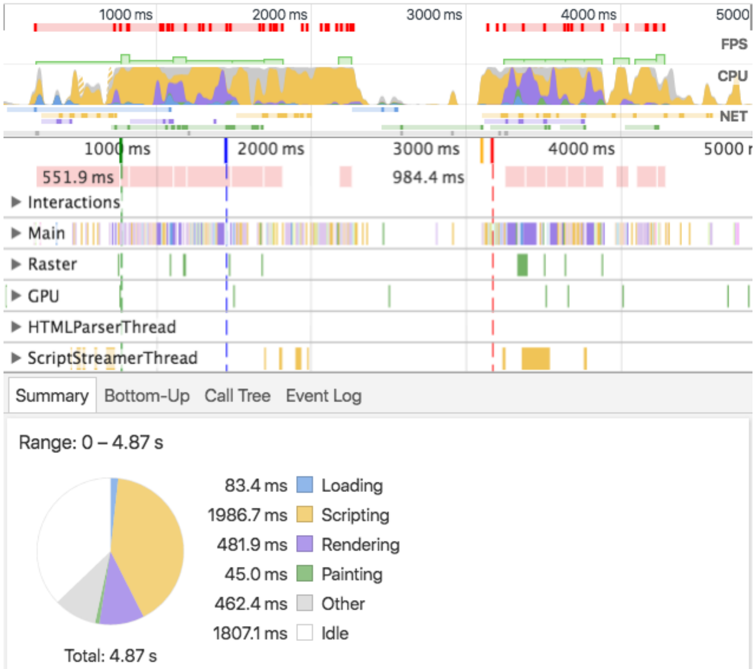
### 1.3.2 Debugging Memory Overhead

All of the browser components in Figure 1.1 have state that contributes to a web application's memory footprint. A higher memory footprint reduces application responsiveness and can even trigger browser tab crashes by exhausting available memory [15, 52, 68, 85, 105]. Browsers contain a few tools that report information about memory usage and memory allocations.

**Heap snapshots**: Some browsers contain a heap snapshot tool that collects a graph of the application's heap, where objects are nodes and references are edges, and displays it to the user. In this paragraph, I refer specifically to views in Google Chrome's heap snapshot tool, which largely matches the functionality in other browsers. Each snapshot includes state from both the JavaScript engine and the layout engine. A summary view groups objects by their type, and displays the total number of objects of that type, their combined shallow size, and their combined retained size. A containment view displays the application's garbage collection roots as trees, letting developers dig down into the application's state. Two snapshots can be diffed to show which objects have been allocated inbetween the snapshots.

**Memory timeline**: In Chrome, the Performance Timeline tool displays basic data associated with the application's memory footprint over time [17]. The timeline contains separate lines for the JavaScript heap size in bytes, the number of active HTML documents, the number of active DOM nodes, the number of active event listeners, and the amount of GPU memory in use.

The JavaScript heap line does not differentiate between live and dead heap state, and merely displays the size of allocated items in the heap. When a garbage collection occurs, the JavaScript heap line drops to reflect the collected garbage. Typically, the memory timeline of a web application describes a sawtooth pattern, where the application's memory usage quickly grows until the JavaScript engine performs a garbage collection that reduces the heap to a baseline size. If the baseline size increases over time, the application might have a leak.



Range: 0 – 4.87 s

| | | |
|---|---|---|
| 83.4 ms | 🟦 | Loading |
| 1986.7 ms | 🟨 | Scripting |
| 481.9 ms | 🟪 | Rendering |
| 45.0 ms | 🟩 | Painting |
| 462.4 ms | ⬜ | Other |
| 1807.1 ms | □ | Idle |

Total: 4.87 s

Figure 1.3: A 5 second timeline collected with Google Chrome while loading Facebook.

### 1.3.3 Understanding Program Behavior

The browser environment is pervasively asynchronous, which makes it challenging for developers to understand their code's behavior. Most browsers contain a performance timeline tool, which provides limited insights into a web application's behavior [7, 17, 86, 98]. Google Chrome's debugger has support for asynchronous call stacks, which extends call stacks into preceding JavaScript events.

**Performance timeline**: Most browsers contain a performance timeline that logs event schedules from individual executions along with other metadata. These timelines present a substantial amount of information over a short duration of time; Figure 1.3 displays a timeline of *less than 5 seconds* collected while loading a Facebook feed. Timelines do not display causal information regarding the source of each event, so developers cannot easily use a timeline to investigate the root causes of asynchronous program behavior.

**Asynchronous call stacks**: Normally, when a JavaScript debugger is paused on a breakpoint, the debugger displays a call stack that ends with the event listener that the browser invoked to handle the current event. Google Chrome's asynchronous call stack feature extends the displayed call stack with the call stack from the JavaScript context that registered the event listener in the first place. From there, the call stack continues to previously recorded asynchronous call stacks.

While asynchronous call stacks provide a causal chain to prior events, they do not support implicit control flow through the DOM and are limited to displaying a single prior async call stack. For example, a JavaScript code may create a new `script` element, assign a URL to its `src` property, and insert it into the webpage, which loads a JavaScript file that initiates a timer. When paused on the timer, the Chrome debugger does not include asynchronous stack traces from the JavaScript code that created the `script` element, breaking the causal chain and preventing developers from examining why the script was loaded in the first place. As a result, asynchronous stack traces provide limited help to developers that are trying to understand complicated program behavior.

### 1.3.4 Porting Code to the Browser

JavaScript is the only programming language that all major web browsers support. Code written in other programming languages can run in the browser via native plugins, or compilers that translate the code into JavaScript.

**Plugins**: Historically, browsers have been able to execute code written in Java, C#/Visual Basic (Silverlight), and ActionScript (Flash) via third-party plugins that execute outside of the browser sandbox and in a native environment [3, 60, 87]. Due to security concerns, all major web browsers will completely disable native plugins by 2020 [44, 65, 92, 117]. The Java, Silverlight, and Flash plugins will be discontinued by that time [2, 118, 124]. As a result, plugins are no longer a viable method for bringing programming languages to the browser.

**Compilers**: One common way to bring programming languages to a new environment is to build a compiler that targets the environment. Compilers exist that translate languages like C and C++ (Emscripten [153]), Java (GWT [51]), and Go (GopherJS [101]) into JavaScript. However, these compilers only support a subset of each language's standard libraries and language features because the browser does not provide direct access to operating system primitives. Developers must manually modify and port any code that depends on missing operating system support. In addition, since the browser does not provide a thread-based execution environment or blocking I/O primitives, most of these compilers require developers to refactor code to execute in an asynchronous and event-driven manner. If the code relies on stack state, developers must manually perform “stack ripping” [4] to convert code into continuation-passing style.

# CHAPTER 2

# CHALLENGES IN WEB DEVELOPMENT

Despite offering a wide variety of conventional development tools, the browser remains a challenging environment for application development. Conventional techniques do not help developers overcome the unique challenges of the browser environment.

## 2.1 Debugging Correctness Issues

Web applications are notoriously frustrating to debug. While browsers contain integrated debuggers, they often provide little assistance to developers. If a bug is the result of a specific event order (a Heisenbug), the act of debugging can disrupt the event schedule and prevent the bug from appearing. Even when a bug does recur while debugging, identifying its root cause can be difficult: in event-driven settings like the web, bug symptoms can manifest far from their root causes.

There are two widely used approaches to find these bugs: scattering logging statements around the program (a.k.a. "printf debugging" using `console.log`), or placing breakpoints to pause the program at specific statements. Both are laborious and iterative processes. Poring over logs to identify bugs often reveals the need to rerun the program with new logging statements in place. Using breakpoints, developers step the program forwards until the first sign that something has gone wrong. Unfortunately, if the breakpoint does not precede the root cause of the bug, the developer must reset breakpoints and restart execution from the beginning. As a result, debugging is currently an arduous and painstaking process for web developers.

## 2.2 Memory Leaks

Browsers have an established reputation for consuming significant amounts of memory [54, 82, 95]. Memory leaks in web applications only exacerbate the situation by further

```
1  class Preview extends PureComponent {
2    // Runs when Preview is added to GUI
3    componentDidMount() {
4      const { codeMirror } = this.props.editor;
5      const wrapper = codeMirror.getWrapperElement();
6      codeMirror.on("scroll", this.onScroll);
7      wrapper.addEventListener("mouseover", this._mover);
8      wrapper.addEventListener("mouseup", this._mup);
9      wrapper.addEventListener("mousedown", this._mdown);
10   }
11 }
```

Figure 2.1: A memory leak in Firefox's debugger. This code (truncated for readability) leaks 0.5MB every time a developer opens a source file. Prior approaches, such as leak detectors that rely on staleness metrics, would fail to find these leaks because the leaked objects (event listeners) are frequently touched.

increasing browser memory footprints. These leaks happen when the application references unneeded state, preventing the garbage collector from collecting it. Web application memory leaks can take many forms, including failing to dispose of unneeded event listeners, repeatedly injecting iframes and CSS files, and failing to call cleanup routines in third-party libraries. Leaks are a serious concern for developers since they lead to higher garbage collection frequency and overhead. They reduce application responsiveness and can even trigger browser tab crashes by exhausting available memory [15, 52, 68, 85, 105].

Prior techniques and existing tools fall short when debugging leaks in web applications. Figure 2.1 displays a representative memory leak in Firefox's debugger, which is a pure HTML5 application that can run as a normal web application in any browser. Lines 6-9 register four event listeners on the debugger's text editor (`codeMirror`) and its GUI object (`wrapper`) every time the user views a source file. The leak occurs because the code fails to remove the listeners when the view is closed. Each event listener leaks `this`, which points to an instance of `Preview`.

### 2.2.1 Prior Automated Techniques

There currently are no effective automated techniques for finding memory leaks in web applications. Previous effective automated techniques for finding memory leaks operate in the context of conventional applications written in C, C++, and Java. These techniques predominantly use a staleness metric to discover [19, 53, 103] or rank [150] memory leaks, but the four memory leaks in the Firefox debugger would not be considered stale. These four listeners continue to execute and touch leaked state every time the user uses the mouse on the editor, marking that state as “fresh”. In web applications, many leaks are connected to browser events.

Prior growth-based techniques assume that leaked objects are uniquely owned (*dominated*) by a single object or that they form strongly connected components in the heap [91, 150]. These assumptions do not hold for the leaked objects in the Firefox debugger because 1) they are owned by four separate leak locations that are only dominated by the global scope, and 2) they reference the global scope (`window`) and are thus strongly connected with nearly the entire heap. These properties are common in web applications.

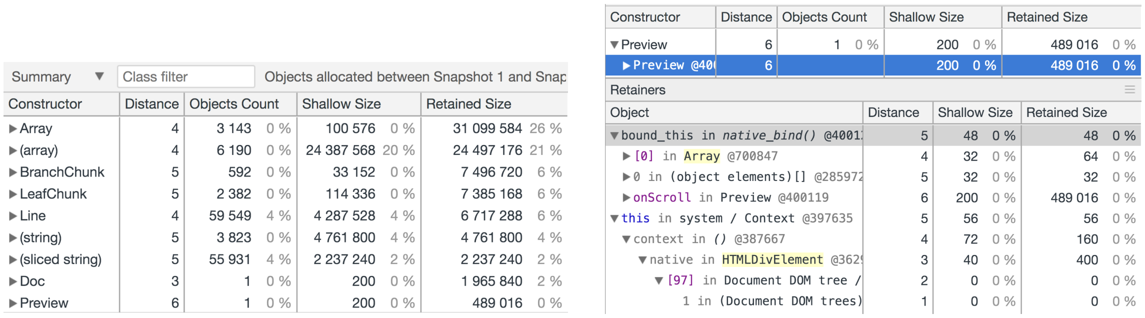
### 2.2.2 Manual Leak Debugging via Heap Snapshots

Since there are currently no automated techniques for identifying memory leaks in web applications, developers are forced to use manual approaches. The current state of the art is manual processing of heap snapshots. This approach does not effectively identify leaking objects or provide useful diagnostic information, and it thus does little to help developers locate and fix memory leaks.

The most popular way to manually debug memory leaks is via the *three heap snapshot technique* introduced by the Gmail team [68]. Developers repeat a task twice on a webpage and examine still-live objects created from the first run of the task. The assumption is that each run will clear out most of the objects created from the previous run and leave behind only leaking objects; in practice, it does not.

To apply this technique to Firefox’s debugger, the developer takes a heap snapshot after loading the debugger, a second snapshot after opening a source file, and a third snapshot after closing and re-opening a source file. Then, the developer filters the third heap snapshot to focus only on objects allocated between the first and second.

This filtered view, shown in Figure 2.2a, does not clearly identify a memory leak. Most of these objects are simply reused from the previous execution of the task and are not actually leaks, but developers must manually inspect these objects before they can come to that conclusion. The top item, `Array`, conflates all arrays in the application under one heading



Summary ▼ Class filter Objects allocated between Snapshot 1 and Snap

| Constructor | Distance | Objects Count | Shallow Size | Retained Size |
|---|---|---|---|---|
| ▶ Array | 4 | 3 143 0 % | 100 576 0 % | 31 099 584 26 % |
| ▶ (array) | 4 | 6 190 0 % | 24 387 568 20 % | 24 497 176 21 % |
| ▶ BranchChunk | 5 | 592 0 % | 33 152 0 % | 7 496 720 6 % |
| ▶ LeafChunk | 5 | 2 382 0 % | 114 336 0 % | 7 385 168 6 % |
| ▶ Line | 4 | 59 549 4 % | 4 287 528 4 % | 6 717 288 6 % |
| ▶ (string) | 5 | 3 823 0 % | 4 761 800 4 % | 4 761 800 4 % |
| ▶ (sliced string) | 5 | 55 931 4 % | 2 237 240 2 % | 2 237 240 2 % |
| ▶ Doc | 3 | 1 0 % | 200 0 % | 1 965 840 2 % |
| ▶ Preview | 6 | 1 0 % | 200 0 % | 489 016 0 % |

(a) A truncated heap snapshot of the Firefox debugger, filtered using the three snapshot technique. The only relevant item is `Preview`, which appears low on the list underneath non-leaking objects.

| Constructor | Distance | Objects Count | Shallow Size | Retained Size |
|---|---|---|---|---|
| ▼ Preview | 6 | 1 0 % | 200 0 % | 489 016 0 % |
| ▶ `Preview @40(` | 6 | | 200 0 % | 489 016 0 % |

Retainers

| Object | Distance | Shallow Size | Retained Size |
|---|---|---|---|
| ▼ `bound_this in native_bind() @4001:` | 5 | 48 0 % | 48 0 % |
| ▶ `[0] in Array @700847` | 4 | 32 0 % | 64 0 % |
| ▶ `0 in (object elements)[] @285972` | 5 | 32 0 % | 32 0 % |
| ▶ `onScroll in Preview @400119` | 6 | 200 0 % | 489 016 0 % |
| ▼ `this in system / Context @397635` | 5 | 56 0 % | 56 0 % |
| ▼ `context in () @387667` | 4 | 72 0 % | 160 0 % |
| ▼ `native in HTMLDivElement @362(` | 3 | 40 0 % | 400 0 % |
| ▼ `[97] in Document DOM tree /` | 2 | 0 0 % | 0 0 % |
| `1 in (Document DOM trees)` | 1 | 0 0 % | 0 0 % |

(b) The retaining paths for `Preview`, the primary leaking object in the Firefox debugger. Finding the code responsible for leaking this object involves searching the entire production code base for identifiers in the retaining paths, which are commonly managed by third-party libraries and obfuscated via minification.

Figure 2.2: The manual memory leak debugging process. Currently, developers debug leaks by first examining heap snapshots to find leaking objects. Then, they try to use retaining paths to locate the code responsible. Unfortunately, these paths have no connection to code, so developers must search their codebase for identifiers referenced in the paths. This process can be time consuming and ultimately fruitless.

because JavaScript is dynamically typed. Confusingly, the entry `(array)` just below it refers to internal V8 arrays, which are not under the application's direct control. Developers would be unlikely to suspect the `Preview` object, the primary leak, because it both appears low on the list and has a small retained size.

Even if a developer identifies a leaking object in a snapshot, it remains challenging to diagnose and fix because the snapshot contains no relation to code. The snapshot only provides retaining paths in the heap, which are often controlled by a third party library or the browser itself. As Figure 2.2b shows, the retaining paths for a leaking `Preview` object stem from an array and an unidentified DOM object. Locating the code responsible for a leak using these retaining paths involves grepping through the code for instances of the identifiers along the path. This task is often further complicated by two factors: (1) the presence of third-party libraries, which must be manually inspected; and (2) the common use of minification, which effectively obfuscates code and heap paths by reducing most variable names and some object properties to single letters.

## 2.3 Understanding Program Behavior

Web applications are completely event-driven, which makes it challenging for developers and automated tools to reason about their behavior. When a web page loads, the HTML and JavaScript comprising the web application register JavaScript functions with the browser to listen to specific events. When loading completes, JavaScript code will only run again in response to listened-for events. This design obscures implicit and asynchronous control flow edges caused by JavaScript interactions or HTML changes that trigger subsequent events.

Existing browser development tools provide limited assistance for understanding a web application's asynchronous behavior. The Chrome debugger's asynchronous call stacks only display a single causal chain leading to prior events, and the chain breaks when an event is caused by a DOM modification. Browser-provided timeline tools log and display a wealth of performance data, but cannot reason about asynchronous control flow. As a result, developers that wish to understand a web application's asynchronous behavior must manually examine the application's code and executions to determine implicit control flow edges.

## 2.4 Porting Code to the Browser

Due to the unique nature of the browser environment, developers are unable to directly compile or translate code written in conventional programming languages into JavaScript for use in a web application. Conventional programming languages and their runtime libraries expect access to primitives for multithreading and synchronous blocking I/O, as well as access to operating system services like a file system, sockets, and processes. However, the browser provides a completely different environment:

- **Single-threaded Execution**: JavaScript is a single-threaded event-driven programming language with no support for interrupts. Events either execute to completion, or until they are killed by the browser because they took too long to finish.

- **Asynchronous-only APIs**: Browsers provide web applications with a rich set of functionality, but emerging APIs are exclusively asynchronous. Due to the limitations of JavaScript, it is not possible to create synchronous APIs from asynchronous APIs.

- **Missing OS Services**: Browsers do not provide applications with access to a file system, socket, or process abstraction. Instead, they offer a panoply of incompatible abstractions.

Developers wishing to port existing code to the web must manually restructure and rewrite the code in JavaScript to adapt to these restrictions.

# CHAPTER 3

# MCFLY: TIME-TRAVEL DEBUGGING FOR THE WEB

Web applications are challenging to debug. Time-traveling debuggers offer the promise of simplifying debugging by letting developers freely step forwards *and backwards* through a program's execution. However, web applications present multiple challenges that make time-travel debugging especially difficult. A time-traveling debugger for web applications must accurately reproduce all network interactions, asynchronous events, and visual states observed during the original execution, both while stepping forwards and backwards. This must all be done in the context of a complex and highly multithreaded browser runtime. At the same time, to be practical, a time-traveling debugger must maintain interactive speeds.

This chapter introduces McFly, the first time-traveling debugger for web applications. McFly departs from previous approaches by operating on a high-level representation of the browser's internal state. This approach lets McFly provide accurate time-travel debugging—maintaining JavaScript and visual state in sync at all times–at interactive speeds. McFly's architecture is browser-agnostic, building on web standards supported by all major browsers. I have implemented McFly as an extension to the Microsoft Edge browser, and core parts of McFly have been integrated into a time-traveling debugger product from Microsoft [89].

## 3.1 McFly

McFly is a prototype time-traveling debugger for web applications. McFly operates on a high-level representation of a browser's internal state, letting it provide accurate time-travel debugging with support for visual state at interactive speeds. This section presents McFly's architecture, which is browser-agnostic.

### 3.1.1 Time-Travel Overview

I first provide an overview of how a developer uses McFly to debug a web application, which structures the remainder of this section.

**Reproducing a bug:** The developer loads the web application with McFly open, and interacts with the application until they discover buggy behavior. While this happens, McFly interacts with the layout engine via a combination of existing DOM interfaces and custom extensions in order to support visual state during time-travel (§3.1.2). Specifically, McFly creates checkpoints of the application's program and visual state (§3.1.3) at a configurable interval (2 seconds by default), and logs I/O and sources of nondeterminism (§3.1.4). By regularly capturing checkpoints, McFly makes it possible to quickly time-travel to an arbitrary point in the web application's execution.

**Debugging:** When the developer encounters a bug, they can place and trigger a breakpoint to begin a debugging session. At this point, the developer can use McFly to step forwards and backwards through the captured program execution to diagnose the bug. To support stepping forwards, McFly uses its log to deterministically replay the program execution.

Stepping backwards is more involved (§3.1.5), as McFly must return the application to a previous state. To go back in time, McFly needs to return the application to a target JavaScript statement $s$ at a specific execution of the statement (at time $t$). To track this information, McFly extends the JavaScript engine with the *branch trace store* and *timestamp store* performance monitors (§3.1.6). McFly uses these performance monitors to determine $s$ and $t$.

**Time-travel:** To time-travel an application to statement $s$ at time $t$, McFly loads the last checkpoint taken before $t$ and replays the log. When execution is at the JavaScript event just prior to $t$, McFly enables the branch trace store and timestamp store, and places a conditional breakpoint on $s$ that triggers at time $t$. Conditional breakpoints are a standard feature supported by all major JavaScript debuggers; the JavaScript engine will only trigger the breakpoint if $s$ executes at time $t$, which completes the time-travel operation.

**Time-travel optimization:** McFly opportunistically generates checkpoints *during replay* to reduce the latency of future time travel operations. In particular, if McFly is time traveling towards $t$, and must start from a "far-away" checkpoint (where distance is defined in terms of JavaScript events), McFly generates a new checkpoint at the JavaScript event just prior to $t$. Thus, a sequence of stepping operations within the same JavaScript event will use the new checkpoint; this is similar to an optimization by Boothe [20].

### 3.1.2 Supporting Visual State

McFly supports visual state during debugging by checkpointing and logging changes to a high-level representation of the layout engine's visual state. The layout engine already reveals much of its internal state in a high-level form to the JavaScript engine via the DOM. However, some of the layout engine's internal state is not accessible via standard interfaces, including the state of animations on the web page and lists of active event listeners. McFly requires access to this state to be able to checkpoint and deterministically re-execute applications.

McFly extends the layout engine with additional debugger-facing interfaces that provide read/write access to a high-level representation of internal layout engine state. These extensions expose the same high-level state described in formal DOM specifications, which all web browsers adhere to [56]. For example, the DOM specification for HTTP request objects (`XMLHttpRequest`) describes network request objects as a state machine; McFly captures these network requests in terms of the internal state machine. As a result, this architecture is *portable* across all major web browsers. Section 4.3 details the specific extensions that the prototype version of McFly supports, as well as their implementation in a widely-used browser.

### 3.1.3 Application Checkpoints

McFly's web application checkpoints contain the application's program state (from the JavaScript engine) and visual state (from the layout engine). As these two types of state are entangled through cross-references, McFly stores them together inside checkpoints as a single object graph.

**Program State:** At a high level, the program state within the JavaScript engine consists of the heap, the stack, and the program counter. However, the JavaScript engine does not maintain a stack or a program counter between JavaScript events. In addition, JavaScript events are typically short-lived (under a few milliseconds in duration) because JavaScript execution blocks UI interactions [112]; long-running events give the user the impression that the application is frozen. This property is enforced by the web browser; if a JavaScript event runs for too long, the browser crashes the tab or raises an alert.

McFly *defers* application checkpoints to occur between JavaScript events. This design constrains where McFly can checkpoint the application, but does not unduly impact debugging performance since most events complete in under a millisecond.

This design ensures that the only program state that McFly needs to capture is the JavaScript heap, which all major web browsers can efficiently traverse using existing garbage collection routines. When the traversal encounters an object in the heap that reflects and retains visual state in the layout engine, McFly serializes its high-level state. To reinstate checkpointed program state, McFly uses internal interfaces present in all major JavaScript engines to reconstruct serialized objects.

**Visual State:** A web application's visual state consists of active GUI nodes on the webpage, inactive GUI nodes stored in the JavaScript heap, active CSS animations, browser-local persistent storage, the state of the random number generator, event listeners, the number of bytes consumed by each active HTML parser, and the state of network requests. McFly serializes a high-level representation of these resources into the checkpoint's object graph using a combination of standard web interfaces and the extensions discussed in Section 3.1.2.

### 3.1.4 I/O and Nondeterminism Log

McFly logs I/O and sources of nondeterminism that it uses to ensure that stepping operations are deterministic. The log contains different types of entries, which each have a different logging and replay strategy. I describe each type of log entry below; Section 4.3 details which DOM interfaces use which strategy.

**Simple entries** correspond to synchronous interactions between JavaScript and the layout engine, such as querying for the date, that depend on browser-external state. McFly logs these values during the original execution, and replays them while debugging.

**Event entries** correspond to JavaScript events. McFly uses these to ensure that JavaScript events occur in the same order as the developer steps through the program. During the original execution, McFly logs a high-level form of each event as it occurs. At debug time, McFly replays events from the log. *This replay strategy reproduces event races observed during the original execution.*

**Inter-event visual state updates** occur between JavaScript events. While JavaScript is executing, the layout engine defers certain visual state updates. For example, layout engines only transition the internal state machine of HTTP request objects in quiescent periods between events. During the original execution, McFly scans for and logs state changes for the relevant high-level state before every JavaScript event. During replay, McFly applies the logged updates before the same event in order to keep the state in sync with JavaScript execution.

**Concurrent visual state updates** occur while JavaScript is executing. The layout engine updates some visual state, such as CSS animations, concurrent with JavaScript execution. Every time JavaScript code synchronously interacts with the layout engine, McFly scans for and logs any changes to concurrently updated state. During replay, McFly prevents the layout engine from concurrently updating state and re-applies the state changes itself during the same synchronous interaction, which keeps the state in sync with JavaScript execution.

McFly uses a counter of synchronous interactions to denote a specific synchronous interaction in the log, and stores the counter value in checkpoints. For example, a log entry with counter value 60 would be applied during the 60th synchronous layout engine interaction, which will be identical across deterministic replays. This strategy preserves any data races between JavaScript code and the layout engine observed during the original execution.

### 3.1.5 Debugger Features

McFly provides a full suite of complements to existing debugger features, and exposes this functionality as an extension to a production JavaScript stepping debugger. I only discuss *step backwards* in this section; the remaining features are implemented similarly.

Step backwards complements *step forwards*, and lets the developer return to the previously-executed program statement. Given that the debugger is paused at the statement $s$ at logical time $t = (c, b)$, where $c$ is the number of times the function has been called since enabling performance monitoring and $b$ is the number of backwards jumps (loop iterations) executed in the current function call, the debugger must determine the statement and logical time of the previously-executed statement, $s'$ and $t'$:

- If $s$ is not the entry point of a basic block, then $s'$ is the previous statement in the block and $t' = t$.
- If $s$ is the entry point of a basic block, then $s'$ is the source statement of the *previously taken branch.*
  - If $s'$ is the current statement in the calling function, then $t'$ is the logical time associated with the caller's call frame.
  - Otherwise, $s'$ is from the same function call as $s$. If $s'$ is a loop header (e.g., `while(someCondition)`), then $s'$ is from a previous loop iteration and $t' = (c, b-1)$. Otherwise, $t' = t$.

Finally, McFly places a conditional breakpoint on $s'$ that triggers when the logical time is $t'$, and replays the program from the previous checkpoint that is closest to the target logical time. If the checkpoint is not close to the target time, McFly records a new checkpoint just before the target JavaScript event in order to speed up subsequent step-back operations.

### 3.1.6 Performance Monitors

To replay execution to a specific statement at a particular point in time, time-traveling debuggers need visibility into the current execution. VM-based time-traveling debuggers typically use performance counters on the processor for this purpose [59, 84], but managed languages, like JavaScript, lack comparable functionality.

McFly augments the JavaScript engine with two performance monitors. The *branch trace store* contains the last branch instruction that was taken by each function that is currently on the call stack. The *timestamp store* contains the timestamp of each function on the call stack. A timestamp is represented as the pair of the function's call count since enabling performance monitoring and the number of backwards jumps (loop iterations) executed thus far in the function call. For example, given the function `function a(){while(true){}}`, the timestamp (3, 2) represents the second iteration of the loop during the third call to the function.

### 3.1.7 Replay Guarantees

McFly guarantees that replay is identical to the original execution, including the data returned from supported layout engine interfaces, the sequence of application-observed JavaScript events, and the pixels on the screen. Animations may not move smoothly during replay, as the system fast-forwards animations to each observed state from the original execution whenever JavaScript code calls into the layout engine, but the JavaScript code will observe the same values seen in the original execution. In addition, developers can use existing debugging tools, such as the DOM Explorer shown in Figure 1.2, to inspect the GUI while debugging.

## 3.2 Implementation

I have implemented a prototype of McFly in the Microsoft Edge web browser. This section describes in detail the changes I made to the layout engine to support McFly's checkpoints and logs (§3.2.1), the changes to the JavaScript engine to support performance monitors (§3.2.2), and the security implications of these changes (§3.2.3).

### 3.2.1 Layout Engine State

Below, I walk through how the prototype implementation of McFly captures the high-level layout engine state described in Table 3.1. Although the table lists many different resources, I only need to make a small number of modifications to the layout engine because standard browser interfaces already make a large amount of high-level state available to the debugger.

**Random numbers:** JavaScript applications use `Math.random()` to generate random numbers, but cannot read or write the internal state of the PRNG. I modify the layout engine to let McFly query and reset the PRNG's state. McFly stores the PRNG state in program checkpoints, and reinstates it prior to replaying from a checkpoint.

| Resource | Interface | High-level State | Logged Data | Log Entries |
|---|---|---|---|---|
| Random Numbers | `Math.random()` | PRNG state | None | |
| Time | `Date` | Internal timestamps | Current time | Simple entries |
| Timers | `setTimeout`, `setInterval` | Active timers | Timer IDs | Simple entries |
| Events | `EventTarget` | Active event listeners | None | |
| | Indirect | None | Sequence of events | Event entries |
| GUI | DOM | Live DOM nodes | Form changes, | Inter-event updates |
| | | | external resource loads, | Concurrent updates |
| | | | HTML parsing progress | Concurrent updates |
| | CSS Animations | Animation status | Animation advancement | Concurrent updates |
| Network Requests | `XMLHttpRequest` | State machine | State machine changes | Inter-event updates |
| Storage | `localStorage`, Cookies | Contents of storage | None | |

Table 3.1: The core browser interfaces that McFly supports. The table above summarizes the different resources that the browser's layout engine provides to JavaScript code, the high-level representation of their internal state, the data that McFly records into its log, and the types of log entries used. In the "interface" column, *Indirect* means that a program's interactions with the resource are implicit, i.e., the interactions do not use an explicit JavaScript interface.

**Current time:** The layout engine contains a `Date` interface that lets programs observe the current time as a `Date` object, which it queries from the OS. I modify the layout engine to let McFly log and replay date requests. To serialize `Date` objects into checkpoints, I use the existing `getTime()` function on the object to retrieve its timestamp.

**Timer status:** JavaScript applications create one-shot timers via `setTimeout()`, and recurring timers via `setInterval()`. Each timer is assigned a unique ID that the layout engine arbitrarily determines. The layout engine does not provide an interface for enumerating the set of active timers or for controlling their IDs. I extend the layout engine to expose the set of active timers and to let McFly log and replay timer ID assignments. I use the former modification to store active timers in checkpoints, and the latter to deterministically replay timer IDs. McFly deterministically replays the timer schedule as discussed under *sequence of events*.

**Events:** JavaScript code can register functions as handlers for events. For example, a program can register handlers for mouse click events. Applications can register event listeners in three ways: through properties on HTML tags (e.g., `<div onclick="a()">`), properties on the DOM elements (e.g., `div.onclick=a;`), or through the `addEventListener()` DOM interface (e.g., `div.addEventListener('click',a)`). JavaScript code can enumerate event listeners registered using the first two approaches, since the listeners are reflected as properties on the associated DOM objects. However, the layout engine does not let JavaScript code enumerate handlers which were registered via `addEventListener()`. Furthermore, the layout engine dispatches events to event handlers in the order in which the handlers were registered, regardless of the registration technique employed. The layout engine does not expose this order, which must be recreated at replay time.

All DOM objects that generate events implement the `EventTarget` interface. I modify the layout engine to let McFly enumerate all event handler information that is associated with an `EventTarget`. McFly uses this extension to store handler orders into checkpoints, and restore handler orders from checkpoints using the preexisting handler registration interfaces.

**Sequence of events:** Each JavaScript execution context is single-threaded and completely event-driven, but the layout engine contains and controls the JavaScript event queue. The layout engine does not expose the queue to JavaScript code, but events must be replayed in the same order as the original execution in order to reproduce event races. I extend the layout engine to let McFly intercept events added to the event queue, which it uses to log and reproduce the original event order during a debugging session.

**GUI:** JavaScript code interacts with the GUI through the DOM tree. Each HTML tag on a web page has a corresponding element object in the tree. Each element object provides JavaScript code with read and write access to tag-specific state, such as the URL for an `<img>` tag or the text in a form field. I use existing JavaScript interfaces to serialize and deserialize the entirety of the DOM tree into checkpoints.

**External resources:** A web page often includes external objects, e.g., HTML tags which specify a `src` attribute and whose content must be fetched from remote servers. The layout engine loads this content in parallel with JavaScript execution on an I/O thread. When the content finishes loading, the layout engine silently updates the applicable HTML element with attributes, such as the `height` and `width` of an image. I modify the layout engine to let McFly log and replay network fetches.

**HTML parsing progress:** A web page contains one or more HTML documents, with secondary HTML documents appearing in frames. The browser incrementally loads and parses HTML documents in parallel with JavaScript execution, which causes new nodes to appear in the DOM. I extend the layout engine to expose the current byte offset in each document's parse stream. McFly logs offset changes during the original execution. During replay, McFly feeds the network stack the new bytes at the appropriate time (via the extension discussed in *external resources*) and waits for the parser to consume them.

**CSS animations:** The DOM does not expose the CSS animation state of an HTML tag—that state resides within the layout engine. If McFly cannot read CSS animation state, then it cannot record an animation's progress with respect to concurrently executing JavaScript code; this would prevent McFly from faithfully recreating the behavior. To enable high-fidelity replays of animations, I modify the layout engine to let McFly read and write the frame counts corresponding to active CSS animations. Using this interface, McFly can "seek" to a specific point in the animation, and keep it synchronized with JavaScript execution.

**Connection status:** JavaScript applications communicate with remote servers via `XMLHttpRequest` objects. Each object encapsulates the state of a single HTTP request. At logging time, the debugger can observe the state of each request, including the content of the HTTP response, using existing methods on `XMLHttpRequest` objects. The debugger needs a mechanism to recreate logged `XMLHttpRequest` objects without creating actual network connections. I extend the layout engine to let the debugger create `XMLHttpRequest` objects from scratch, and set their internal state to arbitrary values.

**Storage:** Web applications manage persistent local data using cookies and the `localStorage` interface [55, 56]. Both mechanisms export a key/value API. The layout engine creates a separate storage area for each origin, and prevents different origins from accessing each other's data. A page's origin is the 3-tuple of the protocol, hostname, and port in the page's URL. Since all of the active origins on a web page execute within the same JavaScript engine, McFly can pose as any origin and manipulate its storage using the same interfaces that are exposed to regular applications.

**Additional browser features:** McFly supports a core set of browser interfaces, which is sufficient to time-travel many existing web applications. Browsers regularly add new features, such as WebGL and Web Audio, that McFly does not support. McFly can be extended to support these features with additional browser modifications to expose their hidden state.

### 3.2.2 Performance Monitors

For simplicity, I implement the *branch trace store* and *timestamp store* by augmenting the browser's JavaScript interpreter. When a performance monitor is enabled, I disable the browser's JIT compiler, forcing JavaScript execution to use the interpreter. As I mention in Section 3.1.1, McFly only requires performance monitoring when a replayed execution nears a target line of interest, so this design has minimal performance impact.

### 3.2.3 Security Implications

The layout engine modifications described in §3.2.1 are only exposed to debugging tools. They are not accessible to web applications via public JavaScript APIs. These modifications do not affect the security model for web content—at logging time and during replay, browsers still use the same-origin policy to isolate content.

## 3.3 Evaluation

I evaluate McFly by running it on a corpus of web applications. The evaluation addresses the following questions:

- **Faithfulness**: Does McFly faithfully and deterministically re-execute web applications?

- **Performance**: Does McFly support time-travel debugging at interactive speeds?

- **Overhead**: Does McFly impose acceptable overhead during normal web application execution?

I performed the evaluation on a desktop with a quad-core Intel Xeon E5-1620 clocked at 3.6 GHz with 16GB of RAM and a 7200 RPM SATA hard drive.

### 3.3.1 Applications

There are no established benchmark suites for time-traveling debuggers for web applications, so I collected one. To perform a controlled evaluation, I chose applications for which I had source code and that I could run locally in a non-production setting. Conducting performance experiments on web applications running in production poses severe methodological challenges. Every experiment is likely to capture a different version of the web application, due to updates or A/B testing, and with different content. For example, Facebook regularly conducts experiments on its users that changes the code and presentation of the website, and Facebook's feed contains different advertisements, posts, and third-party code across visits. While I did not use production applications in this evaluation, I have verified that McFly works on production websites.

I focus the qualitative and quantitative evaluation on benchmarks that exercise different components of McFly:

- **Delta-Blue**, **Earley-Boyer**, **RayTrace**, and **Splay** are from the *Octane* benchmark suite [42], and are memory intensive workloads that stress McFly's checkpoints. I modify the benchmarks to extend their runtime to ∼10 seconds to isolate McFly overhead from parsing/JIT warmup overhead. Unlike the other benchmarks, these programs have no I/O and are deterministic; I exclude these benchmarks from parts of the performance evaluation that use McFly's nondeterminism and I/O log.

- **RayTrace (GUI)** [23] is the RayTrace program from the Octane benchmark suite with its original HTML GUI, which introduces I/O to the program.

- **ColorGame** [62] is an implementation of a test that demonstrates the *Stroop effect* [120]. It uses AngularJS and jQuery, which are both complicated and commonly used libraries, and result in ColorGame having ∼3× as much code as the next largest benchmark. AngularJS exercises a wide variety of DOM features, and encodes crucial application data into the DOM directly. Thus, it is crucial that McFly correctly support this application's visual state during debugging sessions.

- **CRUD** is a standard content management interface that uses jQuery to manage its user interface. CRUD uses HTML forms to let users create, update, and delete content, which McFly must correctly support for debugging to be deterministic.
- **PacMan** [149] is an implementation of the classic Pac-Man game using the HTML5 canvas. It uses timers to update the contents of the canvas every 80ms, and stresses McFly's ability to quickly serialize large DOM objects into checkpoints and log frequent events.

Table 3.2 describes the code size of each of these benchmarks, including HTML documents and JavaScript libraries.

### 3.3.2 Faithfulness

I evaluated the faithfulness of McFly's time-travel debugging by using McFly to debug the benchmark applications. While using breakpoints and McFly's stepping operations, I observed each application's visual and program states, and checked that it matched the original execution.

I manually verified that, across all of the benchmarks, **McFly faithfully and deterministically reproduces the program and visual states observed during the original execution.** Using McFly, I was able to deterministically step forwards and backwards through web application executions while visual state updates, including those induced by CSS animations and network dependencies, remained synchronized with JavaScript execution.

### 3.3.3 Performance

In order to be useful, McFly must step through an execution of a web at interactive speeds. While stepping forwards is straightforward and involves deterministic replay,

| Program | Code | Proc.<br>Size | Create | Write | Checkpoint<br>Resume | Size | Gzipped |
|---|---|---|---|---|---|---|---|
| **Color-Game** | 746 KB | 44 MB | 0.12 s | 0.12 s | 0.43 s | 3.3 MB | 0.9 MB |
| **CRUD** | 250 KB | 28 MB | 0.11 s | 0.10 s | 0.37 s | 2.4 MB | 0.5 MB |
| **Delta-Blue** | 36 KB | 34 MB | 0.09 s | 0.17 s | 0.36 s | 2.1 MB | 0.4 MB |
| **Earley-Boyer** | 244 KB | 123 MB | 0.09 s | 0.11 s | 0.30 s | 3.3 MB | 0.6 MB |
| **PacMan** | 50 KB | 31 MB | 0.13 s | 0.14 s | 0.45 s | 2.2 MB | 0.5 MB |
| **RayTrace** | 38 KB | 73 MB | 0.06 s | 0.09 s | 0.28 s | 2.1 MB | 0.4 MB |
| **Splay** | 17 KB | 538 MB | 0.08 s | 0.10 s | 0.33 s | 2.3 MB | 0.5 MB |
| **Average** | 197 KB | 124 MB | 0.10 s | 0.12 s | 0.36 s | 2.5 MB | 0.5 MB |

Table 3.2: McFly's checkpoint performance. McFly takes a fraction of a second to create or restore a checkpoint. Checkpoints are also significantly smaller than the size of the browser process they capture.

| Program | Overhead | Startup |
|---|---|---|
| **Color-Game** | 4% | 4.5 s |
| **CRUD** | 0% | 3.2 s |
| **PacMan** | 6% | 4.7 s |
| **RayTrace (GUI)** | 5% | 2.9 s |
| **Average** | 4% | 3.8 s |

Table 3.3: McFly's reverse-debugging overhead. Overall, McFly imposes low overhead on the benchmark applications (up to 6% w/ checkpoints every 2 seconds) and requires a short one-time startup cost to initialize efficient reverse-step debugging.

stepping backwards in time involves more costly operations. Specifically, stepping backwards involves resuming execution from the nearest checkpoint and playing forwards to the JavaScript statement of interest. In addition, the first time the developer steps backwards, the debugger creates a checkpoint just prior to the JavaScript event that executes the statement of interest.

McFly's step backwards overhead has two components:

- **Startup Cost**: The first time the developer steps backwards, McFly replays the execution from the nearest checkpoint and creates a new checkpoint just prior to the JavaScript event of interest.

- **Resuming Execution from Checkpoint**: Once the startup cost is paid, the cost of subsequent step backwards operations within the same JavaScript event is dominated by the time to resume from the newly created checkpoint.

I drive each benchmark through a fixed series of events using a PowerShell script that provides application inputs. Each script lasts approximately 10 seconds. I run all benchmark programs and checkpoint their state *every second* for the checkpoint cost evaluation in order to collect more data points, and *every two seconds* for the startup cost evaluation to reflect a more representative value for everyday usage. I calculate the average time to resume from a checkpoint (from disk), and the time to take the first backwards step from 10 random breakpoints. From the results, I observe the following:

**McFly's stepping operations run at interactive speeds in the common case.** After paying a one-time startup fee, the time to execute a backwards step in McFly is dominated by the time it takes to resume from the nearest checkpoint (0.36s on average, as shown in Table 3.2).

**McFly imposes an acceptable backwards step startup cost.** This startup cost is, on average, 3.8 seconds on the benchmark applications or roughly twice the checkpoint rate (Table 3.3).

### 3.3.4 Overhead

To be usable, McFly must not impose significant time and space overheads during normal web application execution. The following metrics contribute to McFly's runtime and space overheads:

- **Log Growth**: The growth rate of the nondeterminism and I/O log. Since a fast-growing log will exhaust disk and memory resources, this metric bounds the practical duration of program executions that McFly can support.

- **Checkpoint Size**: The size of application checkpoints, which McFly takes at a regular and configurable interval during the original execution. If checkpoints are large, then it will be impractical to take frequent checkpoints, which increases the

| **Program** | **Log Growth** |
|---|---:|
| **Color-Game** | 0.6 KB/s |
| **CRUD** | 0.2 KB/s |
| **PacMan** | 1.5 KB/s |
| **RayTrace (GUI)** | 0.9 KB/s |
| **Average** | 0.8 KB/s |

Figure 3.1: McFly's log overhead. McFly's *uncompressed* nondeterminism and I/O log grows slowly.

initial cost return to a specific point in an execution. The *compressed* size indicates the checkpoint's size on disk when compressed with gzip.

- **Checkpoint Creation Time**: The amount of time it takes to create a checkpoint. If it takes a long time to create a checkpoint, then McFly will induce noticeable slowdowns during the initial execution of the program.

To measure these, I again drive each benchmark through a fixed series of events using a PowerShell script for approximately 10 seconds. For checkpoint operations, I run each benchmark in a configuration that takes a checkpoint every second. For log growth, I run each benchmark without taking any checkpoints, maximizing log size. For overall overhead during a normal execution, I measure the runtime of each benchmark in a configuration that takes a checkpoint every two seconds, the default configuration, and compare with the benchmark's runtime without McFly.

As Table 3.2 shows, **checkpoint creation takes less than an eighth of a second** on the benchmark applications. McFly takes an average of 100 milliseconds to create a checkpoint and 120 milliseconds to serialize the checkpoint to disk. McFly's checkpoint operations are fast enough to support frequent checkpoints with acceptable overhead.

**McFly checkpoints compress to less than a megabyte** on the benchmark applications (Table 3.2). McFly checkpoints contain the web application's complete state as a lightweight high-level representation that is amenable to compression. Compressed McFly checkpoints are two orders of magnitude smaller than the browser's memory footprint at the process-level.

**McFly's nondeterminism and I/O log grows at less than 2KB/s** (Figure 3.1). The benchmark with the most nondeterminism, PacMan, has the largest log growth rate of 1.5KB/s. At that rate, McFly could record PacMan's execution for *over 11 years* on a 500GB hard drive.

**With checkpoints every two seconds, McFly's overall overhead is 4% on average** on the benchmark applications (Table 3.3). In some cases, such as CRUD, McFly imposes *no* overhead because checkpoints occur between JavaScript events, when the application is waiting for user input. McFly can impose even lower runtime overheads in exchange for longer step backwards startup costs with a lower checkpoint rate during execution.

## 3.4 Conclusion

This chapter presented McFly, the first time-traveling debugger for web applications. McFly provides accurate time-travel debugging that maintains JavaScript and visual state in sync at all times. I show that McFly lets developers freely step forwards *and backwards* through a web application's execution at interactive speed. Core parts of McFly have been incorporated into a time-traveling debugger product from Microsoft.

# CHAPTER 4

# BLEAK: AUTOMATICALLY DEBUGGING MEMORY LEAKS IN WEB APPLICATIONS

Despite the presence of garbage collection in managed languages like JavaScript, memory leaks remain a serious problem. In the context of web applications, these leaks are especially pervasive and difficult to debug. Web application memory leaks can take many forms, including failing to dispose of unneeded event listeners, repeatedly injecting iframes and CSS files, and failing to call cleanup routines in third-party libraries. Leaks degrade responsiveness by increasing GC frequency and overhead, and can even lead to browser tab crashes by exhausting available memory. Because previous leak detection approaches designed for conventional C, C++ or Java applications are ineffective in the browser environment, tracking down leaks currently requires intensive manual effort by web developers.

This chapter introduces BLeak (**B**rowser **Leak** debugger), the first system for automatically debugging memory leaks in web applications. BLeak leverages the following fact: over a single session, users repeatedly return to the same visual state in modern web sites, such as Facebook, Airbnb, and Gmail. For example, Facebook users repeatedly return to the news feed, Airbnb users repeatedly return to the page listing all properties in a given area, and Gmail users repeatedly return to the inbox view.

I observe that *these round trips can be viewed as an oracle to identify leaks.* Because visits to the same (approximate) visual state should consume roughly the same amount of memory, sustained memory growth between visits is a strong indicator of a memory leak. BLeak builds directly on this observation to find memory leaks in web applications, which (as §7.3 shows) are both widespread and severe.

To use BLeak, a developer provides a short script (17–73 LOC on our benchmarks) to drive a web application in a loop that takes round trips through a specific visual state. BLeak then proceeds automatically, identifying memory leaks, ranking them, and locating their root cause in the source code. BLeak first uses heap differencing to locate locations in the heap with sustained growth between each round trip, which it identifies as leak roots. To directly identify the root causes of growth, BLeak employs JavaScript rewriting to target leak roots and collect stack traces when they grow. Finally, when presenting the results to the developer, BLeak ranks leak roots by return on investment using a novel metric called LeakShare that prioritizes memory leaks that free the most memory with the least effort by dividing the "credit" for retaining a shared leaked object equally among the leak roots that retain them. This ranking focuses developer effort on the most important memory leaks first.

Guided by BLeak, I identify and fix over 50 memory leaks in popular JavaScript libraries and applications including Airbnb, AngularJS, jQuery, Google Analytics, and the Google Maps SDK. BLeak has a median precision of 100% (97% on average). Its precise identification of root causes of leaks makes it relatively straightforward for us to fix nearly all of the leaks I identify (all but one). Fixing these leaks reduces heap growth by 94% on average, saving from 0.5 MB to 8 MB per return trip to the same visual state. I have submitted patches for all of these leaks to the application developers; at the time of writing, 16 have already been accepted and 4 are in the process of code review.

## 4.1 BLeak Overview

This section presents an overview of the techniques BLeak uses to automatically detect, rank, and diagnose memory leaks. I illustrate these by showing how to use BLeak to debug the Firefox memory leak presented in Section 2.2.

**Input script**: Developers provide BLeak with a simple script that drives a web application in a loop through specific visual states. A *visual state* is the resting state of the GUI after the user takes an action, such as clicking on a link or submitting a form. The developer specifies the loop as an array of objects, where each object represents a specific visual state, comprising (1) a `check` function that checks the preconditions for being in that state, and (2) a transition function `next` that interacts with the page to navigate to the next visual state in the loop. The final visual state in the loop array transitions back to the first, forming a loop.

```
1   exports.loop = [
2   // Open a source document
3   {
4     check: function() {
5       const nodes =  $('.node');
6       // No documents are open
7       return $('.source-tab').length === 0 &&
8         // Target doc appears in doc list
9         nodes.length > 1 && nodes[1].innerText === "main.js";
10    },
11    next: function() { $('.node')[1].click(); }
12  },
13  // Close the document after it loads
14  {
15    check: function() {
16      // Contents of main.js are in editor
17      return $('.CodeMirror-line').length > 2 &&
18        // Editor displays tab for main.js
19        $('.source-tab').length === 1 &&
20        // Tab contains a close button
21        $('.close-btn').length === 1;
22    },
23    next: function() { $('.close-btn').click(); }
24  }];
```

Figure 4.1: BLeak input. This script runs the Firefox debugger in a loop, and is the only input BLeak requires to automatically locate memory leaks. For brevity, I modify the script to use jQuery syntax.

Figure 4.1 presents a loop for the Firefox debugger that opens and closes a source file in the debugger's text editor. The first visual state occurs when there are no tabs open in the editor (line 7), and the application has loaded the list of documents in the application it is debugging (line 9); this is the default state of the debugger when it first loads. Once the application is in that first visual state, the loop transitions the application to the second visual state by clicking on `main.js` in the list of documents to open it in the text editor (line 11). The application reaches the second visible state once the debugger displays the contents of `main.js` (line 17). The loop then closes the tab containing `main.js` (line 23), transitioning back to the first visual state.

**Locating leaks**: From this point, BLeak proceeds entirely automatically. BLeak uses the developer-provided script to drive the web application in a loop. Because object instances can change from snapshot to snapshot, BLeak tracks *paths* instead of objects, letting it spot leaks even when a variable or object property is regularly updated with a new and larger object. For example, `history = history.concat(newItems)` overwrites `history` with a new and larger array.

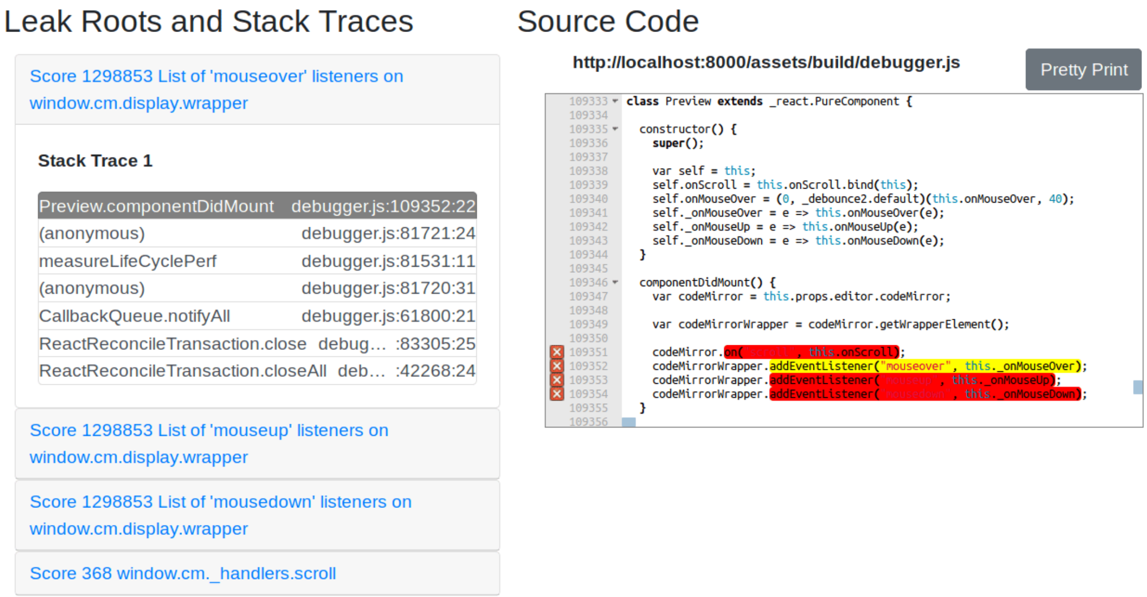
After each visit to the first visual state in the loop, BLeak takes a heap snapshot and tracks specific paths from GC roots that are continually growing. BLeak treats a path as growing if the object identified by that path gains more outgoing references (e.g., when an array expands or when properties are added to an object).

For the Firefox debugger, BLeak notices four heap paths that are growing each round trip: (1) an array within the `codeMirror` object that contains `scroll` event listeners, and internal browser event listener lists for (2) `mouseover`, (3) `mouseup`, and (4) `mousedown` events on the DOM element containing the text editor. Since these objects continue to grow over multiple loop iterations (the default setting is eight), BLeak marks these items as *leak roots* as they appear to be growing without bound.

**Ranking leaks**: BLeak uses the final heap snapshot and the list of leak roots to rank leaks by return on investment using a novel but intuitive metric I call *LeakShare* (§4.2.3) that prioritizes memory leaks that free the most memory with the least effort. LeakShare prunes objects in the graph reachable by non-leak roots, and then splits the credit for remaining objects equally among the leak roots that retain them. Unlike retained size (a standard metric used by all existing heap snapshot tools), which only considers objects *uniquely owned* by leak roots, LeakShare correctly distributes the credit for the leaked `Preview` objects among the four different leak roots since they *all* must be removed to eliminate the leak.

**Diagnosing leaks**: BLeak next reloads the application and uses its proxy to transparently rewrite all of the JavaScript on the page, exposing otherwise-hidden edges in the heap as object properties. BLeak uses JavaScript reflection to instrument identified leak roots to capture stack traces *when they grow* and *when they are overwritten* (not just where they were allocated). With this instrumentation in place, BLeak uses the developer-provided script to run one final iteration of the loop to collect stack traces. These stack traces directly zero in on the code responsible for leak growth.

**Output**: Finally, BLeak outputs its diagnostic report: a ranked list of leak roots (ordered by LeakShare), together with the heap paths that retain them and stack traces responsible



Leak Roots and Stack Traces

Score 1298853 List of 'mouseover' listeners on window.cm.display.wrapper

Stack Trace 1

| | |
|---|---|
| Preview.componentDidMount | debugger.js:109352:22 |
| (anonymous) | debugger.js:81721:24 |
| measureLifeCyclePerf | debugger.js:81531:11 |
| (anonymous) | debugger.js:81720:31 |
| CallbackQueue.notifyAll | debugger.js:61800:21 |
| ReactReconcileTransaction.close | debug… :83305:25 |
| ReactReconcileTransaction.closeAll | deb… :42268:24 |

Score 1298853 List of 'mouseup' listeners on window.cm.display.wrapper

Score 1298853 List of 'mousedown' listeners on window.cm.display.wrapper

Score 368 window.cm._handlers.scroll

Source Code

http://localhost:8000/assets/build/debugger.js

```
class Preview extends _react.PureComponent {

  constructor() {
    super();

    var self = this;
    self.onScroll = this.onScroll.bind(this);
    self.onMouseOver = (0, _debounce2.default)(this.onMouseOver, 40);
    self._onMouseOver = e => this.onMouseOver(e);
    self._onMouseUp = e => this.onMouseUp(e);
    self._onMouseDown = e => this.onMouseDown(e);
  }

  componentDidMount() {
    var codeMirror = this.props.editor.codeMirror;

    var codeMirrorWrapper = codeMirror.getWrapperElement();

    codeMirror.on("scroll", this.onScroll);
    codeMirrorWrapper.addEventListener("mouseover", this._onMouseOver);
    codeMirrorWrapper.addEventListener("mouseup", this._onMouseUp);
    codeMirrorWrapper.addEventListener("mousedown", this._onMouseDown);
  }
```

Figure 4.2: BLEAK output. A snippet from BLEAK's memory leak report for the Firefox debugger. Clicking on a memory leak reveals a list of stack traces, and clicking on a stack frame navigates the source code viewer to the relevant JavaScript statement. BLEAK points directly to the code in Figure 2.1 responsible for the memory leak.

for their growth. BLeak displays this information interactively to the developer via a web application; Figure 4.2 displays a screenshot of BLEAK's output for the Firefox debugger, which points directly to the code responsible for the memory leak from Figure 2.1.

**Summary**: Using BLEAK, the only developer effort required is creating a short script to drive the web application in a loop. BLEAK then locates memory leaks and provides detailed information pointing to the source code responsible. With this information in hand, I was able to discover four new memory leaks in the Firefox debugger, and quickly develop a fix that removes the event listeners when the user closes the document. This fix has been incorporated in the latest version of the debugger.

## 4.2 Algorithms

This section formally describes the operation of BLEAK's core algorithms for detecting (§4.2.1), diagnosing (§4.2.2), and ranking leaks (§4.2.3).

### 4.2.1 Memory Leak Detection

The input to BLeak's memory leak detection algorithm is a set of heap snapshots collected during the same visual state, and the output is a set of *paths* from GC roots that are growing across all snapshots. I call these paths *leak roots*. BLeak considers a path to be *growing* if the object at that path has more outgoing references than it did in the previous snapshot. To make the algorithm tractable, BLeak only considers the shortest path to each specific heap item.

Each heap snapshot contains a heap graph $G = (N, E)$ with a set of nodes $N$ that represent items in the heap, and edges $E$ where each edge $(n_1, n_2, l) \in E$ represents a reference from node $n_1$ to $n_2$ with label $l$. A label $l$ is a tuple containing the type and name of the edge. Each edge's type is either a *closure variable* or an *object property*. An edge's name corresponds to the name of the closure variable or object property. For example, the object `O = { foo: 3 }` has an edge $e$ from `O` to the number 3 with label $l = (\textit{property}, \text{“foo”})$. A path $P$ is simply a list of edges $(e_1, e_2, \ldots, e_n)$ where $e_1$ is an edge from the root node $(G.\textit{root})$.[1]

For the first heap snapshot, BLeak conservatively marks every node as *growing*. For subsequent snapshots, BLeak runs PropagateGrowth (Figure 4.3) to propagate the growth flags from the previous snapshot to the new snapshot, and discards the previous snapshot. On line 2, PropagateGrowth initializes every node in the new graph to *not growing* to prevent spuriously marking new growth as growing in the next run of the algorithm. Since the algorithm only considers paths that are the shortest path to a specific node, it is able to associate growth information with the terminal node which represents a specific path in the heap.

PropagateGrowth runs a breadth-first traversal across shared paths in the two graphs, starting from the root node that contains the global scope (`window`) and the DOM. The algorithm marks a node in the new graph as *growing* if the node at the same path in the previous graph is both growing and has fewer outgoing edges (line 8). As a result, the

[1] For simplicity, I describe heap graphs as having a single root.

```
PropagateGrowth($G, G'$)
 1  $Q = [(G.root, G'.root)]$, $G'.root.mark$ = TRUE
 2  for each node $n \in G'.N$
 3      $n.growing$ = FALSE
 4  while $|Q| > 0$
 5      $(n, n')$ = Dequeue($Q$)
 6      $E_n$ = GetOutgoingEdges($G, n$)
 7      $E'_n$ = GetOutgoingEdges($G', n'$)
 8      $n'.growing = n.growing \wedge |E_n| < |E'_n|$
 9      for each edge $(n_1, n_2, l) \in E_n$
10          for each edge $(n'_1, n'_2, l') \in E'_n$
11              if $l == l'$ and $n'_2.mark$ == FALSE
12                  $n'_2.mark$ = TRUE
13                  Enqueue($(n_2, n'_2)$)
```

Figure 4.3: PropagateGrowth algorithm. PropagateGrowth propagates a node's growth status (*n.growing*) between heap snapshots. BLeak considers a path in the heap to be growing if the node at the path continually increases its number of outgoing edges.

algorithm will only mark a heap path as a leak root if it consistently grows between every snapshot, and if it has been present since the first snapshot.

PropagateGrowth only visits paths shared between the two graphs (line 11). At a given path, the algorithm considers an outgoing edge $e_n$ in the old graph and $e'_n$ in the new graph as equivalent if they have the same label. In other words, the edges have to correspond to the same property name on the object at that path, or a closure variable with the same name captured by the function at that path.

After propagating growth flags to the final heap snapshot, BLeak runs FindLeakPaths (Figure 4.4) to record growing paths in the heap. This traversal visits *edges* in the graph to capture the shortest path to all unique edges that point to growing nodes. For example, if a growing object $O$ is located at `window.O` and as variable $p$ in the function `window.L.z`, FindLeakPaths will report both paths. This property is important for diagnosing leaks, as I discuss in Section 4.2.2.

BLeak takes the output of FindLeakPaths and groups it by the terminal node of each path. Each group corresponds to a specific leak root. This set of leak roots forms the input to the ranking algorithm.

```
FindLeakPaths($G$)
$Q = []$, $T_{Gr} = \{\}$
for each edge $e = (n_1, n_2, l) \in G.E$ where $n_1 == G.root$
    $e.mark$ = TRUE
    Enqueue($Q$, (NIL, $e$))
while $|Q| > 0$
    $t$ = Dequeue($Q$)
    $(t_p, (n_1, n_2, l)) = t$
    if $n_2.growing$ == TRUE
        $T_{Gr} = T_{Gr} \cup \{t\}$
    for each edge $e = (n'_1, n'_2, l') \in G.E$
        if $n'_1 == n_2$ and $e.mark$ == FALSE
            $e.mark$ = TRUE
            Enqueue($Q$, ($t$, $e$))
return $T_{Gr}$
```

Figure 4.4: FindLeakPaths algorithm. FindLeakPaths returns paths through the heap to leaking nodes. The algorithm encodes each path as a list of edges formed by tuples ($t$).

### 4.2.2 Diagnosing Leaks

Given a list of leak roots and, for each root, a list of heap paths that point to the root, BLeak diagnoses leaks through hooks that run whenever the application performs any of the following actions:

- *Grows a leak root* with a new item. This growth occurs when the application adds a property to an object, an element to an array, an event listener to an event target, or a child node to a DOM node. BLeak captures a stack trace, and associates it with the new item.

- *Shrinks a leak root* by removing any of the previously-mentioned items. BLeak removes any stack traces associated with the removed items, as the items are no longer contributing to the leak root's growth.

- *Assigns a new value to a leak root*, which typically occurs when the application copies the state from an old version of the leaking object into a new version. BLeak removes all previously-collected stack traces for the leak root, collects a new stack trace,

```
CalculateLeakShare($G, LR$)
$Q = [G.root]$, $visitId = 0$
for each node $n \in G.N$
    $n.mark = -1$
while $|Q| > 0$
    $n =$ Dequeue($Q$)
    if $n \notin LR$ and $n.mark \neq visitId$
        $n.mark = visitId$
        for each edge $(n_1, n_2, l) \in G.E$ where $n_1 == n$
            Enqueue($Q, n_2$)
for each node $n_{root} \in LR$
    $visitId \mathrel{+}= 1$
    $Q = [n_{root}]$
    while $|Q| > 0$
        $n =$ Dequeue($Q$)
        if $n.mark \neq 0$ and $n.mark \neq visitId$
            $n.mark = visitId$
            $n.counter = n.counter + 1$
            for each $(n_1, n_2, l) \in G.E$ where $n_1 == n$
                Enqueue($Q, n_2$)
for each node $n_{root} \in LR$
    $visitId \mathrel{+}= 1$
    $Q = [n_{root}]$
    while $|Q| > 0$
        $n =$ Dequeue($Q$)
        if $n.counter \neq 0$ and $n.mark \neq visitId$
            $n.mark = visitId$
            $n_{root}.LS \mathrel{+}= n.size/n.counter$
            for each $(n_1, n_2, l) \in G.E$ where $n_1 == n$
                Enqueue($Q, n_2$)
```

Figure 4.5: CalculateLeakShare algorithm. CalculateLeakShare calculates the LeakShare metric ($n.LS$) for a set of leak roots $LR$.

associates it with all of the items in the new value, and inserts the grow and shrink hooks into the new value.

BLeak runs one loop iteration of the application with all hooks installed. This process generates a list of stack traces responsible for growing each leak root.

### 4.2.3 Leak Root Ranking

BLeak uses a new metric to rank leak roots by return on investment that I call *LeakShare* (Figure 4.5). LeakShare prioritizes memory leaks that free the most memory with the least effort by dividing the "credit" for retaining a shared leaked object equally among the leak roots that retain them.

LeakShare first marks all of the items in the heap that are reachable from non-leaks via a breadth-first traversal that stops at leak roots (line 4). These nodes are ignored by subsequent traversals. Then, LeakShare performs a breadth-first traversal from each leak root that increments a counter on all reachable nodes (line 10). Once this process is complete, every node has a counter containing the number of leak roots that can reach it. Finally, the algorithm calculates the LeakShare of each leak root ($n.LS$) by adding up the size of each reachable node divided by its counter, which splits the "credit" for the node among all leak roots that can reach it (line 20).

## 4.3 Implementation

Applying BLeak's algorithms to web applications poses a number of significant engineering challenges:

**Leak identification and ranking**: BLeak uses heap snapshots to identify and rank leaks, but many native methods (implemented in C++) do not expose their state to JavaScript heap snapshots. These native methods can harbor memory leaks and reduce the apparent severity of leaks that retain native state.

**Leak diagnosis**: BLeak's diagnostic strategy assumes that it can collect stack traces when relevant growth occurs, but the browser hides some state and state updates from JavaScript reflection. Native methods bypass JavaScript reflection and mutate state. JavaScript reflection cannot introspect into function closures, necessitating program transformations to expose this state. Transforming a web application is difficult because it can load code at any time from remote servers over HTTP or encrypted HTTPS. In addition, JavaScript contains dynamic features that are necessary but challenging to support with code transformations, including `eval` and `with` statements.

BLeak consists of three main components that work together to overcome these challenges (see Figure 4.6): (1) a driver program orchestrates the leak debugging process (§4.3.1); (2) a proxy transparently performs code rewriting on-the-fly on the target web

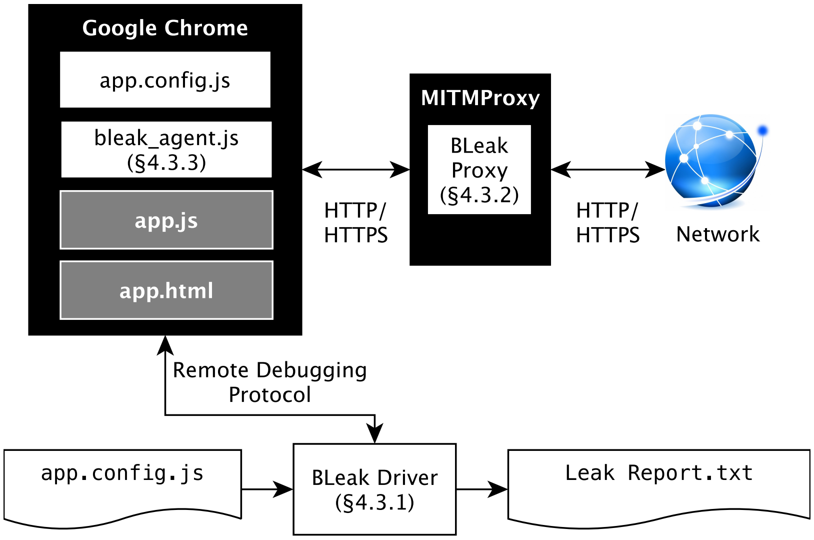


Figure 4.6: BLeak implementation overview. BLeak consists of a driver program that orchestrates the leak detection process (§4.3.1), a proxy that transparently rewrites the target web application's JavaScript during leak diagnosis (§4.3.2), and an agent script embedded in the application that hooks into relevant web APIs and leak roots (§4.3.3). Given a short developer-provided configuration script, BLeak automatically produces a leak report. White rectangles are BLeak components, gray items are automatically rewritten by the proxy during leak diagnosis, and black items are unmodified.

application and `eval`-ed strings (§4.3.2); and (3) an agent script embedded within the application exposes hidden state for leak detection and growth events for leak diagnosis (§4.3.3).

### 4.3.1 BLeak Driver

The BLeak driver is responsible for orchestrating the leak debugging process. To initiate leak debugging, the driver launches BLeak's proxy and a standard version of the Google Chrome browser with an empty cache, a fresh user profile, and a configuration that uses the BLeak proxy. The driver connects to the browser via the standard Chrome DevTools Protocol [47], navigates to the target web application, and uses the developer-provided configuration file to drive the application in a loop. After each repeated visit to the first visual state in the loop, the driver takes a heap snapshot via the remote debugging protocol, and runs PropagateGrowth (Figure 4.3) to propagate growth information between heap snapshots. Prior to taking a heap snapshot, the driver calls a method in the BLeak agent embedded in the web application that prepares the DOM for snapshotting (§4.3.3.2).

At the end of a configurable number of loop iterations (the default is 8), the driver shifts into diagnostic mode. The driver runs FindLeakPaths to locate all of the paths to all of the leak roots (Figure 4.4), configures the proxy to perform code rewriting for diagnosis (§4.3.2), and reloads the page to pull in the transformed version of the web application. The driver runs the application in a single loop iteration before triggering the BLeak agent to insert diagnostic hooks that collect stack traces at all of the paths reported by FindLeakPaths (§4.3.3.1). Then, the driver runs the application in a final loop before retrieving the collected stack traces from the agent. Finally, the driver runs LeakShare (Figure 4.5) to rank the leak roots and generates a memory leak report.

### 4.3.2 BLeak Proxy

The BLeak proxy uses mitmproxy [31] to transparently intercept all HTTP and HTTPS traffic between the web application and the network. The proxy rewrites the web application's JavaScript during leak diagnosis to move closure variables into explicit scope objects, chains scope objects together to enable scope lookup at runtime, and exposes an HTTP endpoint for transforming `eval`-ed code. The proxy also injects the BLeak agent and developer-provided configuration file into the application, uses Babel [11] to translate emerging JavaScript features into code that BLeak can understand, and supports the JavaScript `with` statement. Due to space constraints I do not discuss these features further.

**Exposing closure variables for diagnosis**: During leak diagnosis, the BLeak proxy rewrites the JavaScript on the webpage, including JavaScript inlined into HTML, to make it possible for the BLeak agent to instrument closure variables. Since this process distorts the application's memory footprint, BLeak does not use this process during leak detection and ranking. This code transformation moves local variables into JavaScript "scope" objects (Imagen uses a similar procedure to implement JavaScript heap snapshots [74]). Scope objects are ordinary JavaScript objects where property `foo` refers to the local variable `foo`; the browser-provided `window` object functions as a global scope object, and works identically. BLeak adds a `__scope__` property to all JavaScript `Function` objects that refer to that function's defining scope, and rewrites all variable reads and writes to refer to properties in the scope object. With this transformation, the BLeak agent can capture variable updates in the transformed program in the same manner as object properties.

As an optimization, BLeak performs a conservative escape analysis to avoid transforming variables that are not captured by any function closures. However, if the program calls `eval` or uses the `with` statement, then BLeak assumes that all reachable variables escape.

The scope object transformation treats function arguments differently than local variables. A function's arguments are reflected in an implicit array-like object called `arguments`, and updates to an argument also update the corresponding element in `arguments`.[2] To preserve this behavior, BLeak rewrites updates to arguments so that it simultaneously updates the property in the scope object and the original argument variable.

**Runtime scope lookup**: The JavaScript transformation knows statically which scope objects contain which variables, but the BLeak agent needs this information at runtime to instrument the correct scope object for a given variable. One solution would be to reify scope information into runtime metadata objects that the agent can query, but this would add further runtime and memory overhead. Instead, the proxy uses a simpler design that uses JavaScript's built-in prototype inheritance to naturally encode scope chains. Each scope object inherits from its parent, and the outermost scope object inherits from the browser-provided `window` object. To perform scope lookup, the BLeak agent uses JavaScript reflection to find the first scope object in the chain that defines a property corresponding to the variable.

**`eval` support**: `eval` evaluates a string as code within the context of the call site, posing two key challenges: (1) the string may not be known statically, and (2) the string may refer to outer variables that the code transformation moved into scope objects. The proxy overcomes these challenges by cooperating with the BLeak agent. The proxy transforms all references to `eval` into references to a BLeak agent-provided function that sends the program text synchronously to the proxy for transformation via an HTTP POST. The proxy transforms `eval`-ed code so that references to variables not explicitly defined in the new code refer

[2] This behavior does not occur in "strict mode", but many prominent libraries do not opt into "strict mode".

to a single scope object, and then returns the transformed code to the agent. The agent creates the single scope object as an ECMAScript 2015 Proxy object [100] that interposes on property reads and writes to relay them to the appropriate scope object using runtime scope lookup (Proxy objects are available in modern versions of all major browsers). Finally, the agent calls `eval` on the transformed code. Since this code transformation is independent of calling context, the BLeak agent can cache and re-use transformed code strings.

### 4.3.3 BLeak Agent

The BLeak agent is a JavaScript file that BLeak automatically embeds in the web application; it exposes globally-accessible functions that the BLeak driver can invoke via the Chrome DevTools Protocol. The agent is responsible for installing diagnostic hooks that collect stack traces for growth events. The agent also exposes hidden state in the browser's native methods so that PropagateGrowth (Figure 4.3) can find leaks within or accessible through this state.

#### 4.3.3.1 Diagnostic Hooks

To diagnose memory leaks as described in Section 4.2.2, the BLeak agent needs to interpose on leak root growth, shrinkage, and assignment events. Although all leak roots are JavaScript objects, some types of objects have native browser methods that implicitly grow, shrink, or assign to properties on the object, necessitating interface-specific hooks:

**Object hooks**: BLeak uses Proxy objects to detect when objects gain and lose properties. These Proxy objects wrap JavaScript objects and expose hooks for various object operations, including when the application adds, deletes, reads, or writes properties on the object.

Proxy objects do not automatically take the place of the object they wrap in the heap, so the BLeak agent must replace all references to the object with the proxy to completely capture all growth/shrinkage events. If the agent fails to replace a reference, then it will not capture any object updates that occur through that reference. BLeak can miss a reference if it does not appear in the heap snapshot used for FindLeakPaths (Figure 4.4). This could happen if the heap path to the reference is determined by some external factor, such as the clock, a random number, or the amount of time spent on the page. This behavior appears to be rare; in our evaluation, BLeak reports all but one of the relevant stack traces for all of the true leaks it finds.

Proxy objects are semantically equivalent to the original object except that programs can observe that Proxy($O$) $\neq O$. Since BLeak cannot guarantee that it replaces all references to $O$ with Proxy($O$), a program could run incorrectly if it directly compared these two objects. To preserve correctness, the BLeak proxy also transforms the binary operations `==`, `===`, `!=`, `!==` into calls to an agent function that treats Proxy($O$) as equal to $O$. The BLeak agent also reimplements the functions `Array.indexOf` and `Array.lastIndexOf`, which report the index of a particular item in an array, so that calls with Proxy objects function appropriately.

**Array hooks**: JavaScript arrays contain a number of built-in functions that mutate the array without invoking Proxy hooks. The agent wraps `Array`'s `push`, `pop`, `unshift`, `shift`, and `splice` functions to appropriately capture growth / shrinkage / assignment events.

**DOM node hooks**: Applications can add and remove nodes from the DOM tree via browser-provided interfaces; these operations are not captured via Proxy objects. In order to capture relevant events on DOM nodes, the agent must wrap a number of functions and special properties. On `Node` objects, it wraps `textContent`, `appendChild`, `insertBefore`, `normalize`, `removeChild`, and `replaceChild`. On `Element` objects, it wraps `innerHTML`, `outerHTML`, `insertAdjacentElement`, `insertAdjacentHTML`, `insertAdjacentText`, and `remove`.

**Leak root assignment hooks**: Given a path $P = (e_1, \ldots, e_n)$ to a leak root, the agent instruments all edges $e \in P$ to capture when the program overwrites any objects or variables in the path from the GC root to the leak root. For example, given the path `window.foo.bar`, the program can overwrite `bar` by assigning a new value to `foo` or `bar`. When a leak root gets overwritten with a new value, the agent also wraps that value in a Proxy object.

To interpose on these edges, the agent uses JavaScript reflection to replace object properties with getters and setters that interpose on its modification. Since the BLeak proxy rewrites closure variables into properties on scope objects (§4.3.2), this approach works for all edges in the heap graph.

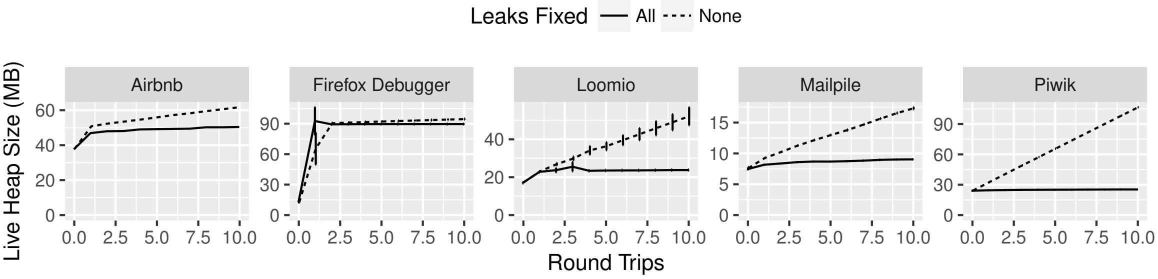


Figure 4.7: Impact of fixing memory leaks found with BLeak. Graphs display live heap size over round trips; error bars indicate the 95% confidence interval. Fixing the reported leaks eliminates an average of 93% of all heap growth.

#### 4.3.3.2 Exposing Hidden State

Some of the browser's native methods hide state from heap snapshots, preventing BLeak from accurately identifying and ranking memory leaks involving this state. To overcome this limitation, the agent builds a mirror of hidden state using JavaScript objects. Using these mirrors, BLeak can locate and diagnose memory leaks that are in or accessible through DOM nodes, event listeners, and partially applied functions.

**DOM nodes**: The agent builds a mirror of the DOM tree as JavaScript objects before the BLeak driver takes a heap snapshot, and installs it at the global variable `$$$DOM$$$`. Each node in the tree contains the array `childNodes` that contains a JavaScript array of (mirror) nodes, and a property `root` that points to the original native DOM node.

**Event listeners**: The agent overwrites `addEventListener` and `removeEventListener` to eagerly maintain an object containing all of the installed listeners. Because this object is maintained eagerly, ordinary object and array hooks capture event listener list growth.

`Function.bind`: The `bind` function provides native support for partial application, and implicitly retains the arguments passed to it. The agent overwrites this function with a pure JavaScript version that retains the arguments as ordinary JavaScript closure variables.

## 4.4 Evaluation

I evaluate BLeak by running it on production web applications. The evaluation addresses the following questions:

- **Precision:** How precise is BLeak's memory leak detection? (§4.4.2)

- **Accuracy of diagnoses:** Does BLeak accurately locate the code responsible for memory leaks? (§4.4.2)
- **Overhead:** Does BLeak impose acceptable overhead? (§4.4.2)
- **Impact of discovered leaks:** How impactful are the memory leaks that BLeak finds? (§4.4.3)
- **Utility of ranking:** Is LeakShare an effective metric for ranking the severity of memory leaks? (§4.4.4)
- **Staleness vs. growth:** How does BLeak compare to a staleness-based leak detector? (§4.4.5)

The evaluation finds **59 distinct memory leaks** across five web applications, *all of which were unknown to application developers.* Of these, 27 corresponded to known-but-unfixed memory leaks in JavaScript library dependencies, of which only 6 were independently diagnosed and had pending fixes. I reported all 32 new memory leaks to the relevant developers along with my fixes; 16 are now fixed, and 4 have fixes in code review. I found new leaks in popular applications and libraries including Airbnb, Angular JS (1.x), Google Maps SDK, Google Tag Manager, and Google Analytics. Appendix A lists each of these memory leaks, the application or library responsible, and links to bug reports with fixes.

I run BLeak on each web application for 8 round trips through specific visual states to produce a BLeak leak report, as in Figure 4.2. I describe these loops using only 17–73 LOC; Appendix B contains the code for each loop. This process takes less than 15 minutes per application on the evaluation machine, a MacBook Pro with a 2.9 GHz Intel Core i5 and 16GB of RAM. For each application, I analyze the reported leaks, write a fix for each true leak, measure the impact of fixing the leaks, and compare LeakShare with alternative ranking metrics.

### 4.4.1 Applications

Because there is no existing corpus of benchmarks for web application memory leak detection, I created one. The corpus consists of five popular web applications that both

| Program | Loop LOC | Leak Roots | False Pos. | Distinct Leaks | Stale Leaks | Prec. | Growth Reduction |
|---|---|---|---|---|---|---|---|
| Airbnb | 17 | 32 | 2 | 32 | 4 | 94% | 1.04 MB (81.0%) |
| Piwik | 32 | 17 | 0 | 11 | 4 | 100% | 8.14 MB (99.3%) |
| Loomio | 73 | 10 | 1 | 9 | 4 | 90% | 2.83 MB (98.3%) |
| Mailpile | 37 | 4 | 0 | 3 | 1 | 100% | 0.80 MB (91.8%) |
| Firefox Debugger | 17 | 4 | 0 | 4 | 0 | 100% | 0.47 MB (98.2%) |
| **Total / mean:** | 35 | 67 | 3 | 59 | 13 | 96.8% | 2.66 MB (93.7%) |

Table 4.1: BLeak precision and accuracy results. On average, BLeak finds these leaks with over 95% precision, and fixing them eliminates over 90% of all heap growth. 77% of these leaks would not be found with a staleness metric (§4.4.5).

comprise large code bases and whose overall memory usage appeared to be growing over time. I primarily focus on open source web applications because it is easier to develop fixes for the original source code; this represents the normal use case for developers. I also include a single closed-source website, Airbnb, to demonstrate BLeak's ability to diagnose websites in production. I present each web application, highlight a selection of the libraries they use, and describe the loop of visual states we use in the evaluation:

**Airbnb [5]**: A website offering short-term rentals and other services, Airbnb uses React, Google Maps SDK, Google Analytics, the Criteo OneTag Loader, and Google Tag Manager. BLeak loops between the pages `/s/all`, which lists all services offered on Airbnb, and `/s/homes`, which lists only homes and rooms for rent.

**Piwik 3.0.2 [109]**: A widely-used open-source analytics platform; I run BLeak on its in-browser dashboard that displays analytics results. The dashboard primarily uses jQuery and AngularJS. BLeak repeatedly visits the main dashboard page, which displays a grid of widgets.

**Loomio 1.8.66 [75]**: An open-source collaborative platform for group decision-making. Loomio uses AngularJS, LokiJS, and Google Tag Manager. BLeak runs Loomio in a loop between a group page, which lists all of the threads in that group, and the first thread listed on that page.

**Mailpile v1.0.0 [77]**: An open-source mail client. Mailpile uses jQuery. BLeak runs Mailpile's demo [76] in a loop that visits the inbox and the first four emails in the inbox (revisiting the inbox in-between emails).

**Firefox Debugger (commit 91f5c63) [38]**: An open-source JavaScript debugger written in React that runs in any web browser. I run the debugger while it is attached to a Firefox instance running Mozilla's SensorWeb [97]. BLeak runs the debugger in a loop that opens and closes SensorWeb's `main.js` in the debugger's text editor.

### 4.4.2 Precision, Accuracy, and Overhead

To determine BLeak's leak detection precision and the accuracy of its diagnoses, I manually check each BLeak-reported leak in the final report to confirm (1) that it is growing without bound and (2) that the stack traces correctly report the code responsible for the growth. To determine BLeak's overhead, I log the runtime of the following specific operations during automatic leak debugging: BLeak's three core algorithms from §4.2 (Algs), proxy transformations from §4.3.2 (Proxy), and receiving and parsing heap snapshots from Google Chrome (Snapshot). I was unable to gather overhead information for Airbnb, the only closed-source application, because the company fixed the leaks I reported prior to this experiment. Table 4.1 summarizes precision and accuracy results, and Table 4.2 summarizes overhead results.

**BLeak has an average precision of 96.8%, and a median precision of 100%** on the evaluation applications. There are only three false positives. All point to an object that continuously grows until some threshold or timeout occurs; developers using BLeak can avoid these false positives by increasing the number of round trips. Two of the three false positives are actually the same object located in the Google Tag Manager JavaScript library.

**With one exception, BLeak accurately identifies the code responsible for all of the true leaks**. BLeak reports stack traces that directly identifies the code responsible for each leak. In cases where multiple independent source locations grow the same leak root, BLeak reports all relevant source locations. For one specific memory leak, BLeak fails to record a stack trace. **Guided by BLeak's leak reports, I was able to fix**

|  | Runtime |  |  |  |
|---|---|---|---|---|
| **Program** | **Total** | **Algs** | **Proxy** | **Snapshot** |
| Piwik | 224 s | 7.1% | 4.7% | 50.3% |
| Loomio | 149 s | 3.7% | 6.2% | 37.9% |
| Mailpile | 388 s | 0.4% | 1.0% | 4.0% |
| Firefox Debugger | 214 s | 2.3% | 37.9% | 33.6% |
| **Mean:** | 243.8 s | 3.4% | 12.5% | 31.4% |

Table 4.2: BLeak overhead results. BLeak takes less than 7 minutes to locate, rank, and diagnose memory leaks in each open-source evaluation application.

**every memory leak.** Fixing each memory leak took approximately 15 minutes. Most fixes involve adding simple cleanup hooks to remove unneeded references or logic to avoid duplicating state every round trip.

**BLeak locates, ranks, and diagnoses memory leaks in less than 7 minutes** on the open-source evaluation applications. BLeak's core algorithms (PropagateGrowth, FindLeakPaths, CalculateLeakShare) contribute less than 8% to that runtime. The primary contribution to overhead on all benchmarks, with one exception, is receiving and parsing Chrome's JSON-based heap snapshots. The Firefox Debugger spends more time in the proxy because it uses new JavaScript features that BLeak supports by invoking the Babel compiler, which dominates proxy runtime for that application [11].

### 4.4.3 Leak Impact

To determine the impact of the memory leaks that BLeak reports, I measure each application's live heap size over 10 loop iterations with and without my fixes. I use BLeak's HTTP/HTTPS proxy to directly inject memory leak fixes into the application, which lets us test fixes on closed-source websites like Airbnb. I run each application except Airbnb 5 times in each configuration (I run Airbnb only once per configuration for reasons discussed in §4.4.4).

To calculate the leaks' combined impact on overall heap growth, I calculate the average live heap growth between loop iterations with and without the fixes in place, and take the difference (Growth Reduction). For this metric, I ignore the first five loop iterations because these are noisy due to application startup. Figure 4.7 and Table 4.1 present the results.

**Growth Reduction for Top Leaks Fixed**

| Program | Metric | 25% | 50% | 75% |
|---|---|---|---|---|
| Airbnb | LeakShare | 0K | 111K | **462K** |
| | Retained Size | 0K | 0K | 105K |
| | Trans. Closure Size | 0K | **196K** | 393K |
| Loomio | LeakShare | 0K | **1083K** | **2878K** |
| | Retained Size | **64K** | 186K | **2898K** |
| | Trans. Closure Size | 59K | 67K | 2398K |
| Mailpile | LeakShare | **613K** | **817K** | **820K** |
| | Retained Size | **613K** | **817K** | **820K** |
| | Trans. Closure Size | 0K | 0K | 201K |
| Piwik | LeakShare | **8003K** | **8104K** | **8306K** |
| | Retained Size | 2073K | 7969K | **8235K** |
| | Trans. Closure Size | 103K | 110K | 374K |

Table 4.3: Performance of memory leak ranking metrics. The table displays growth reduction by metric after fixing quartiles of top ranked leaks. **Bold** indicates greatest reduction (±1%). I omit Firefox because it has only four leaks which must all be fixed (see §2.2). LeakShare generally outperforms or matches other metrics.

**On average, fixing the memory leaks that BLeak reports eliminates over 93% of all heap growth** on the evaluation applications (median: 98.2%). These results suggest that BLeak does not miss any significantly impactful leaks.

### 4.4.4 LeakShare Effectiveness

I compare LeakShare against two alternative ranking metrics: retained size and transitive closure size. Retained size corresponds to the amount of memory the garbage collector would reclaim if the leak root were removed from the heap graph, and is the metric that standard heap snapshot viewers display to the developer [64,85,96,105]. The transitive closure size of a leak root is the size of all objects reachable from the leak root; Xu *et al.* use this metric along with staleness to rank Java container memory leaks [150]. Since JavaScript heaps are highly connected and frequently contain references to the global scope, I expect this metric to report similar values for most leaks.

I measure the effectiveness of each ranking metric by calculating the growth reduction (as in §4.4.3) over the application with no fixes after fixing each memory leak in ranked order. I then calculate the quartiles of this data, indicating how much heap growth is eliminated after fixing the top 25%, 50%, and 75% of memory leaks reported ranked by a given metric. I sought to write patches for each evaluation application that fix a single leak root at a time, but this is not feasible in all cases. Specifically, one Airbnb patch fixes two leak roots; one Mailpile patch (a jQuery bug) fixes two leak roots; and one Piwik patch, which targeted a loop, fixes nine leak roots. In these cases, I apply the patch during a ranking for the first relevant leak root reported.

I run each application except Airbnb for ten loop iterations over five runs for each unique combination of metric and number of top-ranked leak roots to fix. I avoid running duplicate configurations when multiple metrics report the same ranking. Airbnb is challenging to evaluate because it has 30 leak roots, randomly performs A/B tests between runs, and periodically updates its minified codebase in ways that break my memory leak fixes. As a result, I was only able to gather one run of data for Airbnb for each unique configuration. Table 4.3 displays the results.

**In most cases, LeakShare outperforms or ties the other metrics.** LeakShare initially is outperformed by other metrics on Airbnb and Loomio because it prioritizes leak roots that share significant state with other leak roots. Retained size always prioritizes leak roots that uniquely own the most state, which provide the most growth reduction in the short term. LeakShare eventually surpasses the other metrics on these two applications as it fixes the final leak roots holding on to shared state.

### 4.4.5 Leak Staleness

I manually analyzed the leaks BLeak finds to determine whether they would also be found using a staleness-based technique. I assume that, to avoid falsely reporting most event listeners as stale, a staleness-based technique would exercise each event listener on the page that could be triggered via normal user interaction. In this case, no memory leaks stemming from event listener lists would be found by a staleness-based tool. Leaks in internal application arrays and objects that emulate event listener lists for user-triggered events would also not be found. Finally, I assume that active DOM elements in the DOM tree would not be marked stale, since they are clearly in use by the webpage. Memory leaks stemming from node lists in the DOM would also not be found by a staleness-based technique. **Of the memory leaks BLeak finds, at least 77% would not be found with a staleness-based approach.** Table 4.1 presents results per application (see Appendix A for individual leaks).

## 4.5 Conclusion

This chapter presented BLeak, the first effective system for debugging client-side memory leaks in web applications. I show that BLeak has high precision and finds numerous previously-unknown memory leaks in production web applications and libraries. BLeak is open source [145], and is available for download at `http://bleak-detector.org/`.

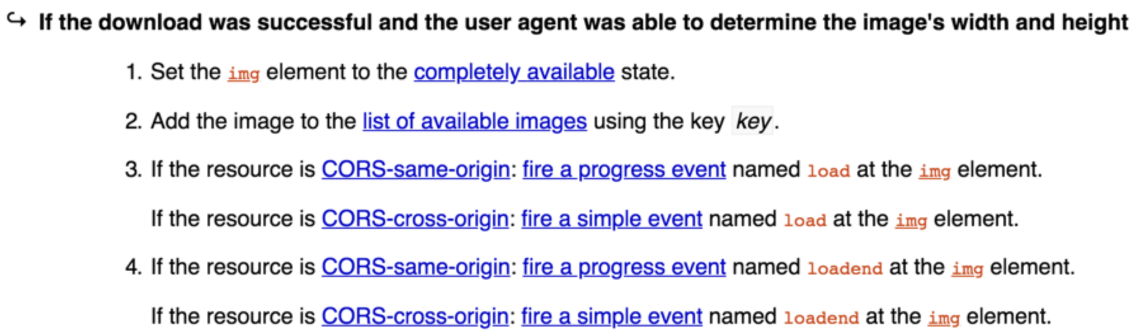
# CHAPTER 5

# BCAUSE: CAUSAL PROGRAM UNDERSTANDING FOR WEB APPLICATIONS

Web applications are challenging to understand because they are pervasively asynchronous. After a web application initializes, JavaScript code only executes in response to browser events, which break application execution across thousands of short-duration periods. It is challenging to track a web application's control flow through these events, as each event's root causes are not always apparent.

This chapter introduces BCause, a framework for understanding asynchronous control flow in web applications. During a browsing session, BCause records a trace of JavaScript events and implicit control flow edges between JavaScript events. Once collected, BCause can use the trace to reason about web application behavior and construct a *causal graph* of the events in the execution. The causal graph is a weakly connected directed graph; every node in the graph, which correspond to JavaScript events, can be reached from the root node, which corresponds to the root HTML document of the web application, via edges that correspond to asynchronous control flow edges. Thus, BCause can reason about the *root causes* of every event in the execution by tracing a path back to the root node.

I demonstrate the accuracy and utility of BCause's graphs by building AdBlame, which uses the causal graph to predict the bandwidth consumption of a web application with ad blocking enabled to *within 2.2% of the ground truth.* AdBlame uses BCause's tracing infrastructure to log program actions that may initiate network requests, and analyzes the resulting causal graph to filter out requests that depend on blocked scripts and HTML documents. On a corpus of production websites, I show that AdBlame is significantly more accurate at predicting bandwidth savings than directly applying ad blocker filter rules to a network traffic log.



↪ **If the download was successful and the user agent was able to determine the image's width and height**

1. Set the `img` element to the completely available state.
2. Add the image to the list of available images using the key *key*.
3. If the resource is CORS-same-origin: fire a progress event named `load` at the `img` element.

   If the resource is CORS-cross-origin: fire a simple event named `load` at the `img` element.
4. If the resource is CORS-same-origin: fire a progress event named `loadend` at the `img` element.

   If the resource is CORS-cross-origin: fire a simple event named `loadend` at the `img` element.

Figure 5.1: Snippet from the HTML5 standard that describes events on Image objects [146]. Setting the `src` property on an Image object causes the browser to download the image; if the download succeeds, then the browser triggers `load` and `loadend` events. All JavaScript events are specified informally in standards documents like this one.

## 5.1 Challenges

There are two primary challenges to building a causal graph of a web application's events: 1) determining the program actions that precede specific events, and 2) collecting sufficient information from a program's execution to construct the graph.

**Specifying happens-before relations**: In order to build a causal graph of a web application's events, it is necessary to understand what program actions may initiate (and *happen-before*) specific JavaScript events. If event $E_1$ performs program action $A$, and $A$ happens-before event $E_2$, then there should be a link in the causal graph between $E_1$ and $E_2$. For example, setting the `src` property on an image object initiates a network request to fetch the image, which causes a `load` event to occur. In addition, before the web application can observe the `load` event, it must register a JavaScript function to listen for the event via `addEventListener` or the `onload` property. Thus, setting `src` and registering an event listener on an image *happen-before* `load`.

Unfortunately, these happens-before relations are only informally specified in plain-English standards documents. Figure 5.1 displays a snippet from the HTML5 standard that describes when the `load` event occurs on Image objects. Previous systems that require knowledge of event causality, including JavaScript race detectors [106, 113] and tools for understanding program behavior [6, 72, 73], manually encode every happens-before relation needed to support their use cases. As discussed further in Section 8.3, these systems miss many happens-before relations. For example, these systems do not recognize that setting the `src` property on an Image object causes the `load` event to occur, but do recognize that the program must register an event listener for `load`.

**Collecting execution traces**: Constructing a causal graph also requires collecting sufficient information about a program's execution. Many previous systems that reason about event causality modify the web browser itself to collect this information [72,106,113], which reduces the overhead of logging but limits portability across browsers and browser versions. Other systems require no browser modifications, and instead use JavaScript reflection to interpose on select browser interfaces [6,73]. All of these systems contain manually written logging logic for each of the happens-before relations that they support, which is a fragile and error-prone approach.

## 5.2 BCause Overview

This section provides an overview of how BCause supports identifying happens-before relations between program actions and events, collects an execution trace, and uses the trace to produce a causal graph.

**Specifying happens-before relations**: While previous systems rely on fragile hand written happens-before logic, BCause takes a novel approach that automates the process of inferring happens-before rules and generating logging code. BCause automatically classifies browser interfaces into five distinct categories (§5.3) using WebIDL (§5.4.3) [81]. BCause uses this classification to automatically infer most happens-before relations and to generate JavaScript code that runs alongside the web application and logs causal information for each type of interface. Since modern web browsers use WebIDL to generate bindings between their native C++ codebases and JavaScript, BCause can rely upon browser implementations of web standards matching the interface specification. Using browser-supplied WebIDL files, BCause will automatically support new web interfaces as browsers implement them.

When a happens-before relation is not evident from static type information, such as the relation between the `src` property of an Image and the `load` event, I manually annotate the relevant WebIDL file with the relation. Annotations are a well-supported feature of

WebIDL, and browsers use them to inform code generation. BCause uses these happens-before annotations to automatically generate the required logging logic.

**Collecting a trace**: To collect a trace, the developer opens a web browser configured to use BCause, and navigates to their web application. While the developer interacts with the web application, BCause logs important web application actions to disk, including:

- The start and end of every JavaScript event.
- The start and end of HTML and JavaScript initialization.
- Program actions that may initiate (and *happen before*) JavaScript events, HTML initialization, and JavaScript initialization. These program actions include event listener registrations *and* HTML/JavaScript code that cause events to occur. Each program action includes a stack trace that points to the JavaScript statements responsible for the action.

Since JavaScript execution is single-threaded (i.e., triggered events wait for prior events to finish executing), these trace entries are totally ordered by the sequence in which they occur. BCause ends the trace when the developer closes the browser or navigates to a different URL.

**Constructing a causal graph**: Given a trace file, BCause can construct a causal graph of a web application's events. Each node in the graph represents a JavaScript event or the initialization of an HTML document. An edge from node A to node B indicates a program action that happened during node A that caused node B to later occur, and contains a stack trace pointing to the JavaScript code that caused the action to occur.

Using this graph, developers and automated tools can determine why a program is performing certain actions, even when control flow edges are obscured by DOM interactions. Figure 5.2 displays a subgraph of eBay's behavior collected with BCause. The graph displays the chain of activities that cause eBay to create an IFrame containing HTML from the advertising service `stags.bluekai.com`:

1. The HTML of `https://www.ebay.com/` contains an inline script on line 288. Inline scripts execute in order as the HTML document loads.

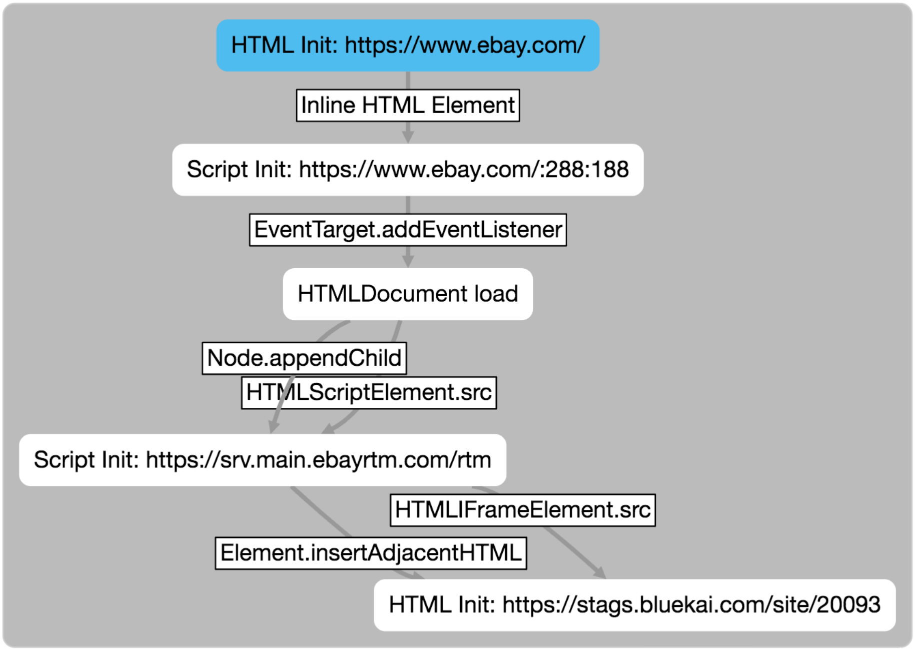

HTML Init: https://www.ebay.com/
Inline HTML Element
Script Init: https://www.ebay.com/:288:188
EventTarget.addEventListener
HTMLDocument load
Node.appendChild
HTMLScriptElement.src
Script Init: https://srv.main.ebayrtm.com/rtm
HTMLIFrameElement.src
Element.insertAdjacentHTML
HTML Init: https://stags.bluekai.com/site/20093

Figure 5.2: A subgraph of eBay.com's causal graph collected with BCause. The graph displays the chain of program actions that causes eBay to create an IFrame that points to `stags.bluekai.com`, an advertising service.

2. The inline script adds an event listener on the HTML document's `load` event, which executes once the HTML document and its resources finish loading.

3. The event listener inserts a script element into the DOM via `appendChild`, and points it to `https://srv.main.ebayrtm.com/rtm` by writing the `src` attribute. This action causes the browser to fetch and execute the JavaScript code at that URL.

4. When `rtm` initializes, it injects a string containing HTML into the DOM via `insertAdjacentHTML`, which contains an IFrame. The code then points the IFrame to `https://stags.bluekai.com/site/20093` by writing the `src` attribute. The browser then fetches and executes the HTML at that URL.

Since each edge contains a full stack trace, BCause points developers directly to the lines of code responsible for each event.

## 5.3 Trace Entries

In order to build a causal graph, BCause collects a trace of web application actions at runtime. The trace comprises start and end entries for JavaScript events and HTML initializations, along with program actions that may cause future events to occur. BCause needs to be able to associate each JavaScript event, HTML initialization, and JavaScript initialization with the one or more program actions that caused it. Accordingly, trace entries refer to a specific JavaScript event or initialization event using unique identifiers that are tied to specific objects or browser API invocations. This section describes how BCause trace entries refer to each type of major program event. Later, Section 5.4.3 describes how BCause uses WebIDL to automatically recognize browser interfaces that correspond to each major program event and generate appropriate logging logic.

### 5.3.1 DOM Events

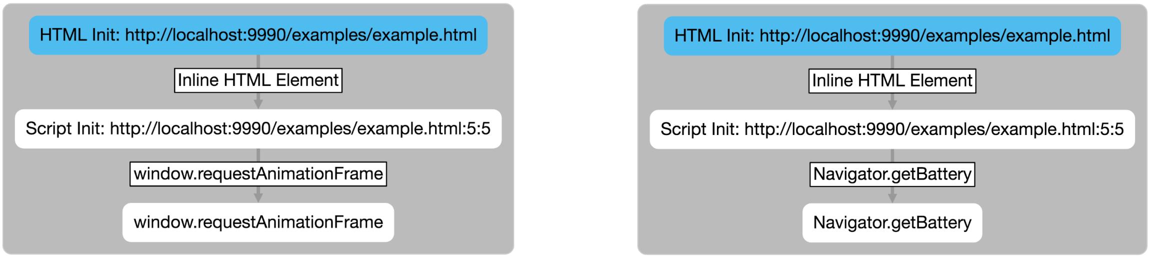
Certain DOM objects are *event targets* that emit named DOM events. For example, DOM nodes emit user interaction events including “click”, “keydown”, and “scroll”. A web application can register a JavaScript function as an *event listener* for one or more of these events via `addEventListener` or event handler properties that begin with `on`. The browser will invoke the event listener when the listened-for events occur.

BCause associates each event target object with a unique ID, and each event listener *registration* with a unique ID. Subtly, a JavaScript function registered as an event listener multiple times will have a unique ID for each registration. At the beginning and end of an event listener’s execution, BCause creates a trace entry that contains both IDs along with the name of the event. BCause also creates a trace entry with the same identifying information when event listeners are registered and unregistered, which appear as incoming edges to event listener execution nodes in the causal graph.

When the program performs an action that may cause a DOM event on an event target, BCause creates a log entry that contains the tuple of the event target ID and event name. These log entries do not refer to specific event listeners, as each program action precedes the execution of *any* event listener for a specific event on a specific object. For example, if the web application sets the `src` property of an HTML script element, the browser will

| **Object Type** | **Properties** | **Action** | **Causes** |
|---|---|---|---|
| Element | scroll{, To, By} | Call | scroll |
| Element | scroll{Top, Left} | Set | scroll |
| FileReader | readAs{ArrayBuffer, BinaryString, Text, DataURL} | Call | error, load{, start, end}, progress |
| FileReader | abort | Call | abort |
| FileWriter | write, truncate | Call | error, progress, write{, end, start} |
| FileWriter | abort | Call | abort |
| HTMLElement | focus | Call | selectionchange (on `this.ownerDocument`) |
| HTMLIFrameElement | src, srcdoc | Set | error, load, HTML init. |
| HTMLIFrameElement | N/A | Add to DOM | HTML init. |
| HTMLImageElement | src | Set | error, load, loadend |
| HTMLLinkElement | href | Set | error, load |
| HTMLMediaElement | src | Set | error, load{, start, end} |
| HTMLScriptElement | src | Set | error, load, JavaScript init. |
| HTMLScriptElement | text | Set | JavaScript init. |
| HTMLScriptElement | N/A | Add to DOM | JavaScript init. |
| WebSocket | constructor | Construct | close, error, message, open |
| WebSocket | close | Call | close |
| Window | scroll{ , To, By} | Call | scroll (on `this.document`) |
| XMLHttpRequest | send | Call | error, load{, end, start}, progress, timeout, upload (on `this` and `this.upload`); readystatechange (on `this`) |

Table 5.1: Hand-annotated happens-before relations for standard web APIs. All additional relations are automatically derived from browser-supplied WebIDL files. *Causes* contains the set of events that the action may cause. Some properties on element object types, such as `src`, are reflected as HTML attributes; BCause tracks both HTML attribute and JavaScript property changes. For space reasons, the table excludes annotations on nine interfaces related to the IndexedDB object database.


HTML Init: http://localhost:9990/examples/example.html
Inline HTML Element
Script Init: http://localhost:9990/examples/example.html:5:5
window.requestAnimationFrame
window.requestAnimationFrame
HTML Init: http://localhost:9990/examples/example.html
Inline HTML Element
Script Init: http://localhost:9990/examples/example.html:5:5
Navigator.getBattery
Navigator.getBattery

(a) Causal graph of a program that calls the callback API `requestAnimationFrame`.

(b) Causal graph of a program that calls the Promise API `Navigator.getBattery`.

Figure 5.3: Causal graphs of programs that invoke different types of asynchronous operation APIs. BCause accurately captures causal graphs for both callback-based and Promise-based asynchronous operations.

either emit a `load` event or an `error` event on the script element depending on if it could load the script. If event listeners for either event executes after this occurs, then BCause knows to draw a link in the causal graph between the node that performed the action and nodes corresponding to each event listener execution. I manually identified these special program actions, as the connection between events and specific program actions are only described informally in web standards documents. Table 5.1 contains a listing of manually identified program actions that precede or cause events.

### 5.3.2 Asynchronous Operations

Some browser APIs correspond to asynchronous operations that the browser performs on behalf of the web application. The web application provides the API with JavaScript functions, which the browser invokes once the operation completes or fails. The browser contains two different types of asynchronous operation APIs: callback-based and Promise-based.

**Callbacks**: Callback APIs accept one or more JavaScript functions as arguments, which are later invoked zero or more times as an operation proceeds. Timers created via `setTimeout` and `setInterval` fall under this category, as does `requestAnimationFrame`, which invokes a callback before the next time the GUI is repainted.

BCause associates each invocation of a callback API with a unique invocation ID. When the web application invokes a callback API, BCause logs a trace entry containing this invocation ID. When the browser later invokes a function passed to the callback API, BCause logs start and end trace entries that refer to the original invocation of the callback API by its invocation ID. With this information, BCause constructs a link in the causal graph between the graph node that invoked the callback API and nodes corresponding to invocations of functions passed to the API. Figure 5.3a displays an example causal graph for a program that calls `requestAnimationFrame`.

**Promises**: Emerging asynchronous browser APIs return a Promise object. A Promise object encapsulates the state and result of an asynchronous operation. Promises are a recent addition to the browser, and were only standardized in ECMAScript 2015. Web applications can use methods on the Promise object to register JavaScript functions that run when the Promise resolves successfully or is rejected with an error message. Promises can also be chained together so that values and errors flow through multiple Promise objects, which obscures the causal links between program actions and Promise events.

To support Promises without directly tracking their complicated internal control flow, BCause wraps each browser-provided Promise in a pure JavaScript implementation of a Promise. BCause traces the JavaScript Promise implementation alongside the application, which ensures that BCause's causal graphs properly track causal links through Promise objects without any special Promise support. When the web application invokes an API that returns a Promise, BCause logs a trace entry with a unique ID corresponding to the API invocation. When the browser resolves or rejects the Promise, BCause logs a trace entry with the same ID, which forms an edge in the causal graph between the node that invoked the API and the JavaScript functions that the resolved or rejected Promise invokes. Figure 5.3b displays an example causal graph for a program that calls the Promise-based API `Navigator.getBattery`.

### 5.3.3 Cross-document Messages

Using the `postMessage` interface on Window and MessagePort objects, JavaScript code can send messages and certain types of objects to other documents within the web application. BCause modifies each message to contain a unique ID, which it logs to track the event that created the message. When the web application processes a `message` event, BCause reads the ID from the message and includes it in a trace entry that marks the start and end of a message event. It then removes the ID from the message before sending it on to the application.

### 5.3.4 JavaScript Initialization

A web application can include JavaScript code via `script` HTML elements in the DOM. This code executes asynchronously when its contents load. *JavaScript initialization* occurs when a script element's JavaScript code executes for the first time. Before a script element can initialize JavaScript code, it must be inserted into the DOM tree, and it must either contain inline JavaScript source code or its `src` property must refer to a URL containing JavaScript.

BCause associates each script element with a unique ID, and logs a trace entry containing the ID when any precondition to script initialization occurs (see Table 5.1). At the start and end of JavaScript initialization, BCause logs a trace entry referring to the same ID, along with a second trace entry specifying that the current initialization precedes a `load` DOM event on the script element. Figure 5.2 shows a causal graph of eBay.com that contains JavaScript initialization for `https://srv.main.ebayrtm.com/rtm`.

### 5.3.5 HTML Initialization

A web application's execution begins with the browser loading and initializing a root HTML document specified in the browser's address bar. It is necessary to track HTML initialization as it directly influences JavaScript execution. HTML can contain inline JavaScript event listeners, HTML attributes that initiate JavaScript events (corresponding to some properties in Table 5.1), `script` elements that initialize JavaScript code, and `iframe` elements that embed other HTML documents.

BCause log entries identify the root HTML document by a unique ID associated with the root `document` object, and subdocuments by a unique ID associated with each IFrame object. BCause creates a log entry that refers to this ID every time the web application creates an IFrame object or changes its source document, which forms an incoming edge to the document's HTML initialization node in the causal graph. When an HTML document initializes, BCause emits trace entries for any HTML content that may initiate subsequent events bookended by trace entries indicating the start and end of trace entries associated with this specific HTML document's initialization.

JavaScript code can inject additional HTML snippets into the webpage as a string via interfaces like `innerHTML` and `document.write`. BCause does not emit start and end events for snippets, but does emit log entries for any injected HTML content that may initiate JavaScript events. In the causal graph, these log entries appear as outgoing edges from the node that injected the HTML.

## 5.4 Implementation

As shown in Figure 5.4, BCause comprises four components: 1) the BCause proxy injects the agent and initialization hooks into JavaScript and HTML files, 2) the BCause server collects traces, 3) the BCause hook generator automatically generates JavaScript code that intercepts web application interactions with the browser, and 4) the BCause agent runs alongside the web application and sends trace events to the server. During trace generation, the developer uses an unmodified Google Chrome web browser that is configured to use the BCause proxy. To facilitate JavaScript hooks into privileged browser APIs, BCause disables several browser security features with command line arguments, including the same origin policy and origin isolation within separate processes. BCause could operate around these security restrictions with further browser integration. BCause also disables service workers, which can perform application-specific caching services by interposing on network requests. BCause could be extended to support service workers by tracking happens-before relations through requests processed by the worker.

### 5.4.1 BCause Proxy

The BCause proxy uses mitmproxy [31] to transparently intercept all HTTP and HTTPS traffic between the web application and the network. The proxy rewrites HTML and JavaScript documents to inject the BCause agent into the application, and to call into the agent to create trace entries that log the start and end of HTML and JavaScript initialization.

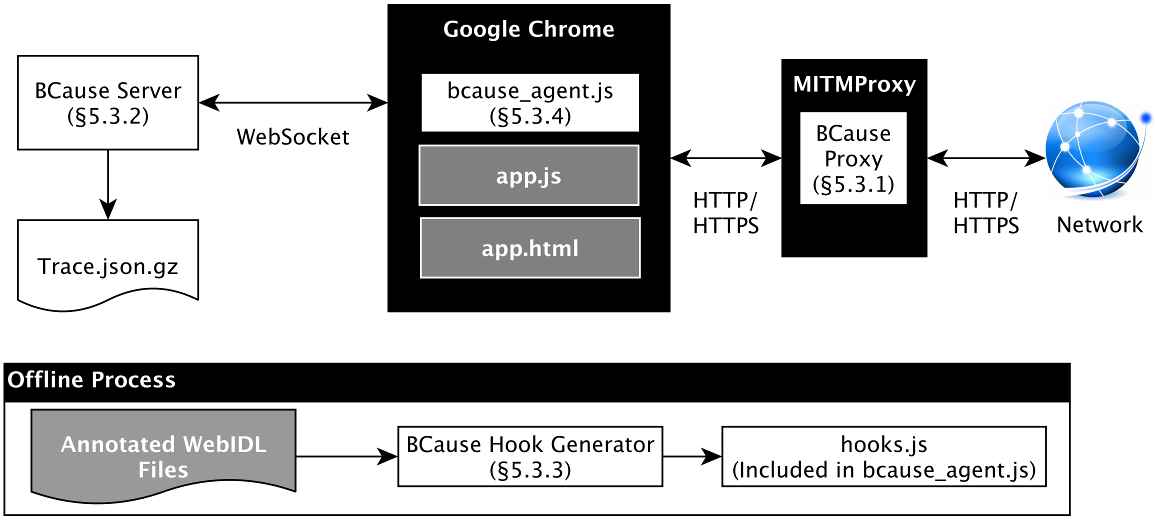


Figure 5.4: BCause implementation overview. BCause consists of a proxy that rewrites HTML and JavaScript (§5.4.1), a server that stores trace files (§5.4.2), a hook generator that automatically generates shims that intercept web application interactions with browser APIs (§5.4.3), and an agent that monitors the DOM for changes and sends trace events to the server (§5.4.4). These components work together to trace a web application's execution in an unmodified Chrome browser.

### 5.4.2 BCause Server

The BCause server accepts WebSocket connections from the BCause agent. Each WebSocket connection corresponds to a single trace; the server pipes all messages from the BCause agent into a gzipped trace file on disk.

### 5.4.3 BCause Hook Generator

Web standards documents use WebIDL, or Web Interface Description Language, to define new interfaces between JavaScript and the browser. Many modern web browsers use WebIDL to generate bindings between their native C++ codebases and JavaScript; this process ensures that their implementations of web APIs hew closely to the standard.

I annotate WebIDL files from Chromium's source code with the happens-before relations listed in Table 5.1. The BCause hook generator parses these annotated WebIDL files using the webidl2 library [147], and generates a JavaScript file that installs hooks into the browser environment. Each hook replaces a browser-provided method or object property with a version that calls into the BCause agent to produce the trace entries described in Section 5.3.

The hook generator automatically infers the remaining hooks using information already present in Chromium's WebIDL files:

**Asynchronous operations and DOM events**: Since WebIDL is statically typed, the hook generator automatically finds and generates hooks for asynchronous operations and event listener properties (e.g., `onclick` on HTMLElement objects). Callback-based asynchronous operations accept a function as an argument, promise-based asynchronous operations return a Promise object, and event listener properties are of type `EventHandler`.

**Reflected HTML attributes**: Some JavaScript properties from Table 5.1 are actively reflected as HTML attributes; the web application can change the property's value via JavaScript and HTML. For example, a web application can set the `src` property on an HTMLScriptElement in the following ways: 1) as a JavaScript property (`script.src = newSrc`), 2) as an HTML attribute (`script.setAttribute("src", newSrc)`), and 3) within an HTML document (`<script src="newSrc"></script>`). BCause must support all three variants in order to accurately track event causality.

Chromium's WebIDL files annotate JavaScript properties that reflect an HTML attribute with the `Reflect` WebIDL attribute. The BCause hook generator uses this information to determine which HTML attributes to monitor using the BCause agent.

**Cross-document messages**: Web applications can send messages across documents via the `postMessage` function on Window and MessagePort objects. Chromium's WebIDL files contain a `PostMessage` annotation on these functions, which guides BCause to this type of function.

### 5.4.4 BCause Agent

The BCause agent runs alongside the web application and has four primary duties: 1) sending trace entries to the BCause server, 2) exposing hooks that monitor for DOM changes, 3) hiding BCause's modifications to the web application's HTML and JavaScript code from JavaScript reflection APIs, and 4) rewriting dynamically injected HTML and JavaScript. Since every HTML document maintains a separate copy of the browser's APIs, the root document and every subdocument within an `iframe` contain a copy of the agent.

Excluding library dependencies and autogenerated hooks, the source code to the BCause agent is approximately 1.6 KLOC.

**Sending trace entries**: The BCause agent contains functions that send different types of trace entries to the BCause server. The hooks generated by the hook generator call into these functions to log trace entries. These functions assign or look up the unique IDs for the trace entries described in Section 5.3 and send the entry to the BCause server via a WebSocket connection. All of the BCause agents on the webpage share a single WebSocket connection; BCause agents contained within IFrames send their trace entries to the root agent for forwarding to the server. Thus, trace entries are stored in the order in which they occurred at runtime.

**DOM hooks**: The BCause agent instruments the DOM APIs responsible for manipulating HTML elements, and exposes a set of hooks for monitoring for changed attributes and inserted HTML elements. The BCause hook generator uses these DOM hooks to log trace entries for attribute changes and element insertions that may initiate JavaScript events or register JavaScript event handlers.

**Hiding code modifications**: Web applications sometimes make assumptions about the content of the HTML and JavaScript on the webpage. BCause's HTML and JavaScript modifications may invalidate some of these assumptions, causing application errors. For example, AngularJS [46] applications encode HTML templates directly into the HTML of the document, and dynamically fill in each template with information at runtime. Some applications also inject `script` elements containing objects in JavaScript Object Notation (JSON) format that the application later reads and parses from the DOM.

The BCause agent instruments core DOM APIs to hide any HTML and JavaScript modifications that it or the proxy makes. When the application queries the DOM for a list of HTML elements, BCause returns a modified list that does not contain any BCause-injected elements. When the application serializes HTML via APIs like `innerHTML`, BCause parses the HTML, removes BCause modifications, and serializes the result as a string. If the application reads the code contained within a `script` element, the BCause agent removes any injected calls to the agent. The BCause agent also removes changes to HTML and JavaScript documents when the web application manually downloads them via `Xml-`

HttpRequest or fetch, as the proxy blindly rewrites all HTML and JavaScript documents. As a result, web applications executing under BCause can continue to make assumptions about the content of their HTML and JavaScript code.

**Rewriting dynamically injected content**: Although the BCause proxy rewrites JavaScript and HTML files requested from the network, it is unable to rewrite dynamically generated JavaScript and HTML content. Web applications can create and inject new script and iframe elements with JavaScript or HTML code specified as a string or encoded within a data: URL, causing new initialization events that BCause needs to track.

The BCause agent performs the same rewriting actions as the BCause proxy on dynamically generated HTML and JavaScript. The agent uses the same rewriting code as the proxy, since the proxy is also written in JavaScript.

## 5.5 Evaluation

In this section, I evaluate BCause in two dimensions:

- **Accuracy**: As a case study to measure BCause's accuracy, I build a tool called AdBlame that uses BCause's causal graphs to predict the impact of ad blocking on a web application's bandwidth consumption.
- **Overhead**: I measure BCause's runtime overhead on Speedometer 2.0 [21], a browser benchmark suite comprised of GUI-driven web applications.

### 5.5.1 Accuracy

There is no established benchmark suite for web application event causality, and, since happens-before relations are only informally specified, there is no way to automatically determine ground truth. Thus, it is infeasible to directly calculate the accuracy of BCause's causal graphs. Instead, I indirectly measure BCause's accuracy by building a tool for which ground truth is available and then calculating its accuracy.

Specifically, I build a tool called AdBlame that uses BCause's causal graph to predict a web application's bandwidth consumption with ad blocking enabled. Ad blockers contain comprehensive community-maintained filter lists that determine which network requests

| **Object Type** | **Properties** | **Action** | **Notes** |
|---|---|---|---|
| HTMLElement | style | Set | Contains CSS as text; scans for uses of URL() |
| HTMLIFrameElement | src | Set | |
| HTMLImageElement | src | Set | |
| HTMLImageElement | srcset | Set | Contains list of images |
| HTMLLinkElement | href | Set | Uses proxy to intercept and scan CSS contents for uses of URL() |
| HTMLMediaElement | N/A | Add to DOM | URL read via `currentSrc` or `src` |
| HTMLScriptElement | src | Set | |
| HTMLStyleElement | N/A | Add to DOM | Contains CSS as text; scans for uses of URL() |
| Window | fetch | Call | |
| XMLHttpRequest | send | Call | |

Table 5.2: Program actions that initiate network requests. AdBlame uses BCause's existing hook infrastructure to log when these program actions occur. While this list is not exhaustive, it is sufficient to accurately predict the bandwidth consumption of production websites. Using BCause, AdBlame can easily install additional hooks to support additional interfaces.

should be blocked. AdBlame uses BCause's causal graph and these filter lists to determine which network requests depend on blocked content. In order to track network requests, AdBlame augments BCause's traces with *network request* entries that log program actions that trigger network requests. AdBlame uses BCause's existing hook infrastructure to intercept the program actions listed in Table 5.2 that may initiate network requests, and to associate them with nodes in the causal graph. It then combines the causal graph with a network log from the proxy, which contains the time and size of each network request on the wire, to estimate the bandwidth consumption of a web application execution with ad blocking enabled.

To measure the utility of the causal information for this particular application, I compare AdBlame against an approach that runs the filter list directly on the network log. I expect that this straightforward approach will not work well; filter lists are intended to block advertising network scripts that the page directly embeds, and not all URLs that serve advertising content. In addition, rules in a filter list can depend on the context of a network request, such as whether it is a script element, subdocument (IFrame), or CSS stylesheet.

|  | **Bandwidth** |  | **Prediction (% Difference)** |  |
|---|---|---|---|---|
| **Website** | **Ads** | **No Ads** | **AdBlame** | **Network Filter** |
| CNN.com | 4749 KB | 3379 KB | 3304 KB (2.2%) | 4606 KB (36.3%) |
| Dictionary.com | 1904 KB | 515 KB | 515 KB (0.0%) | 1024 KB (98.8%) |
| Weather.com | 5159 KB | 3789 KB | 3868 KB (2.1%) | 4428 KB (16.9%) |

Table 5.3: AdBlame's accuracy on our benchmark web sites. AdBlame's predictions are within 2.2% of the ground truth. In contrast, directly applying the filter list to the network log of each visit is significantly less accurate.

I infer the context as best I can with the network log using the MIME type specified in the header of each HTTP response, although I note that servers do not always specify the correct MIME type. I map HTML MIME types to subdocuments, JavaScript MIME types to scripts, CSS MIME types to stylesheet, image MIME types to image, and the rest to no particular context.

To measure the ground truth, I load each webpage in Chrome with uBlock Origin and use Chrome's built-in developer tools to record bandwidth consumption [41]. I configure uBlock, BCause, and the baseline approach to use two community maintained filter lists, EasyList and EasyPrivacy [36], as they are comprehensive and portable across ad blockers and filter list parsers. For benchmarks, I chose three major websites that contain a significant number of advertisements: CNN.com, Dictionary.com, and Weather.com. I visit the front page of each site twice, with BCause and with uBlock, and wait for all graphical content within the browser viewport to finish loading. Both predictions and the bandwidth consumption of the web page with ads are derived from data collected during BCause's run. Table 5.3 contains the results of this experiment.

**AdBlame's predictions are within 2.2% of the ground truth.** The difference from ground truth can be explained by natural deviations in the content of each web application between visits, network requests initiated by program actions not included in Table 5.2, and slight differences in web application behavior between the two runs caused by disabled browser features like ServiceWorkers. In contrast, directly applying the filter lists to the network log results in predictions that are, on average, within 50% of the ground truth.

These results suggest that BCause's causal graphs are accurate, as they were necessary to produce accurate bandwidth consumption predictions.

### 5.5.2 Overhead

To evaluate the runtime overhead of our BCause prototype, I run the Speedometer 2.0 benchmark suite in the Chromium browser with and without BCause [21]. Speedometer repeatedly loads different to-do list implementations in an `iframe`. This behavior is nearly a worst-case scenario for BCause, as each `iframe` reload causes a new BCause agent to instantiate and install hooks for the new document. I also note that I have not spent any time optimizing BCause to improve performance, as our primary goal is producing accurate models of web application behavior. I run the experiment in Chromium 68.0.3419.0 on an 2015 MacBook Pro with a 2.9 GHz Intel Core i5 and 16GB of RAM.

**Overall, BCause imposes a 2.58X slowdown on Speedometer 2.0.** Much of this overhead is due to initialization overhead. I believe this is acceptable overhead for a debugging tool.

## 5.6 Conclusion

This chapter presented BCause, a framework for understanding a web application's asynchronous behavior. BCause produces a trace of a web application's execution, and uses that trace to build a causal graph of its events. BCause requires no browser changes, and uses existing WebIDL files to automatically determine most causal links between events. I demonstrate the accuracy of BCause's causal graphs by building AdBlame, which predicts the impact of ad blocking policies on bandwidth consumption to within 2.2% on a suite of production web applications.

# CHAPTER 6

# DOPPIO: BREAKING THE BROWSER LANGUAGE BARRIER

Developers are unable to reuse code written in conventional programming languages in the browser because the browser does not provide the environment that these languages expect. The browser lacks synchronous blocking I/O, multiple threads of execution, and traditional operating system services. As a result, directly translating the code into JavaScript is not generally possible.

This chapter identifies and describes how to resolve the impedance mismatch between the browser and the native environment that conventional programming languages expect. I present Doppio, a runtime system that makes it possible to execute unmodified applications written in conventional programming languages inside the browser [143]. Its execution environment overcomes the limitations of the JavaScript single-threaded event-driven runtime model by providing language implementations with emulated threads that support suspending and resuming execution to enable blocking I/O and multithreading in the source language. To support standard library and language features, Doppio provides common POSIX-like operating system abstractions including a file system abstraction, network sockets, and an unmanaged heap for dynamic memory allocation.

I demonstrate the feasibility of using Doppio through two case studies. I present the DoppioJVM, a prototype yet robust implementation of a Java Virtual Machine interpreter on top of Doppio that can run complex unmodified JVM programs in the browser without plugin support. I show that the combination of Doppio and the DoppioJVM makes it possible to run full, unmodified JVM applications inside a wide range of browsers. I also demonstrate Doppio's generality by augmenting an existing C/C++ to JavaScript compiler, Emscripten [153], with Doppio and present a case study of porting a C++ game to the browser.

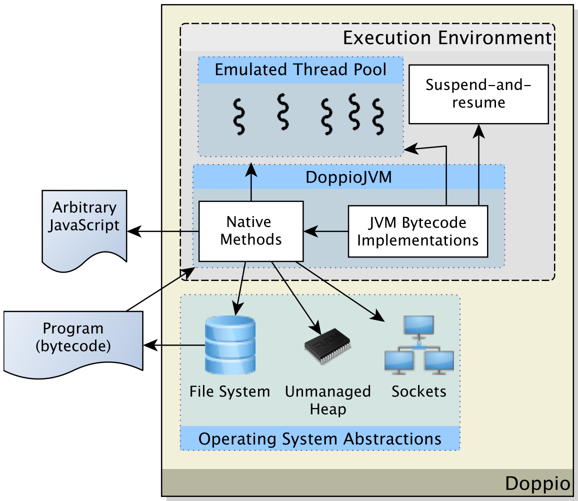


Figure 6.1: A diagram of the DOPPIO runtime system. The DOPPIO runtime system makes it possible for existing programs to execute in an unmodified browser via its virtualized execution environment and operating system abstractions. This diagram displays the various components of DOPPIO at a high level, and illustrates how language implementations, such as DOPPIOJVM, rely on them.

## 6.1 Execution Environment

In this section, I describe how DOPPIO's entirely JavaScript-based execution environment automatically segments existing programs into finite-duration events to keep the web page responsive, emulates synchronous APIs in the *source* language in terms of asynchronous JavaScript APIs, and implements multithreading.

### 6.1.1 Automatic Event Segmentation

To cope with the browser's execution model, DOPPIO must break up the execution of existing programs into finite-duration events. To perform this task, DOPPIO's execution environment contains a mechanism called *suspend-and-resume* that allows an executing program to suspend itself to the heap to be resumed later, letting other events in the browser event queue like user input execute. Because this mechanism is not natively available in JavaScript, languages implemented using DOPPIO must satisfy two properties:

**The call stack must be explicitly stored in JavaScript objects.** JavaScript lacks comprehensive introspection APIs and has no mechanism for saving stack state. As a result, programs executing in DOPPIO can only reliably use the JavaScript stack for transient state that will not be needed for program resumption.

**The program must be augmented to periodically check if it should suspend.** JavaScript lacks preemption: once an event starts executing, it will continue executing until it completes or is killed by the browser. As a result, a language implemented using DOPPIO must call the execution environment periodically to check if it should suspend execution to free up the JavaScript thread.

With an explicit call stack representation in hand, the DOPPIO execution environment can suspend a program for later resumption. To do so, it first creates an anonymous function—the *resumption callback*— that captures the call stack in a closure and that contains the logic needed to resume the program. It then passes the function to an asynchronous browser mechanism such as `setImmediate` that will invoke it later. Finally, it notifies the language implementation that it should halt execution, with a promise that it will handle resuming the program from that point later.

To prevent applications from executing for too long, DOPPIO uses a counter to determine when an application needs to suspend. Each suspend check initiated by the language implementation decrements the counter by 1; when the counter reaches 0, the application needs to suspend. DOPPIO resets the counter to an appropriate value calculated using a cumulative moving average of how often the program checks the counter.

### 6.1.2 Emulating Blocking with Asynchronous APIs

Using a variant of suspend-and-resume, DOPPIO makes it possible to emulate a synchronous API in the source language in terms of an asynchronous JavaScript API. When it wishes to invoke an asynchronous JavaScript function, the language implementation must craft a callback function that encapsulates the logic for migrating the data provided through the asynchronous API into items that the language can understand. DOPPIO wraps this callback in a variation of the resumption callback, and then calls the asynchronous API with the modified callback function. When the browser triggers the resumption callback, the program executing in DOPPIO resumes as if it had just received data synchronously from a regular function call in its language.

### 6.1.3 Multithreading Support

Doppio implements multithreading by exploiting the fact that programs executing in Doppio maintain an explicit representation of their stack. Since JavaScript lacks a mechanism for preempting execution, multithreading is necessarily cooperative from the JavaScript point of view. However, as language implementations must voluntarily specify valid context switches to Doppio, the semantics of multithreading may be preemptive in the source language.

Doppio provides language implementations with a mechanism for switching threads, which is a variation of the suspend-and-resume functionality. Doppio maintains a "thread pool" – essentially an array of call stacks. When the language implementation determines that it is time for a context switch, Doppio saves the call stack of the currently running thread into this array, and chooses another thread to resume. Language implementations can provide a scheduling function that determines which thread to resume. By default, Doppio uses a weighted round robin scheduling policy.

## 6.2 OS Services

The web browser lacks a number of core operating system features that existing programs depend on, such as the file system, access to unmanaged memory, and network sockets. As a result, these abstractions must be implemented in terms of the resources available in the browser so that arbitrary programs can run in the web environment. This section outlines how Doppio implements these abstractions.

### 6.2.1 File System

Browsers provide a hodgepodge of persistent storage mechanisms with different storage formats, restrictions, compatibility across browsers, and intended use cases. Many do not expose synchronous interfaces, making it impossible to implement a blocking file system on top of them.

However, by combining Doppio's thread emulation with a unified asynchronous file-based storage abstraction, Doppio can provide existing programs with the synchronous file system semantics they expect, with high compatibility across browsers. Doppio's filesystem has two primary components: (1) an implementation of the Node.js file system API, which is a light JavaScript wrapper around POSIX-like file system calls, and (2) a backend API for defining different “file system” backends for each persistent storage solution. Doppio contains backends for six separate file storage mechanisms, including browser-local storage, zip files, and Dropbox cloud storage.

### 6.2.2 Unmanaged Heap

Programs use the unmanaged heap either to perform unsafe memory operations (in managed languages), or as the source of dynamically allocated memory (in unmanaged languages). Doppio emulates the unmanaged heap using a straightforward first-fit memory allocator that operates on JavaScript arrays. Data stored to and read from Doppio’s heap are actually copied; updates must be kept in sync according to the language’s semantics.

### 6.2.3 TCP Sockets

For security reasons, browsers do not provide JavaScript applications with direct access to network sockets. Instead, browsers provide a feature called WebSockets that enable JavaScript applications to make *outgoing* full-duplex TCP connections with WebSocket servers. JavaScript applications cannot accept *incoming* WebSocket connections.

WebSocket connections use a custom handshake and data frame format. However, existing socket-based servers and clients expect a standard TCP handshake and the ability to define custom application-layer data frame formats. As a result, they will not be able to send or receive WebSocket connections out of the box.

Resolving this problem requires a solution for *clients* running in the browser that make outgoing socket connections, and *servers* running on native hardware that expect incoming socket connections. Doppio resolves the client side of the issue by emulating a Unix socket API in terms of WebSocket functionality. The freely-available Websockify program provides a solution for the server end of the problem; it wraps unmodified programs, and translates incoming WebSocket connections into normal TCP connections [78].

## 6.3 DoppioJVM

To demonstrate Doppio's suitability as a full-featured operating environment for executing unaltered applications written in conventional programming languages, I built DoppioJVM. DoppioJVM is a robust prototype Java Virtual Machine (JVM) interpreter that operates entirely in JavaScript. This section describes a number of DoppioJVM's key features, and how they rely on support provided by Doppio.

### 6.3.1 Segmented Execution

Due to the JavaScript execution model, DoppioJVM must execute as finite-duration events to prevent the browser from stopping its execution. DoppioJVM uses Doppio's suspend-and-resume functionality to achieve this. However, it must satisfy the requirements outlined in Section 6.1.1 before it can use this mechanism.

DoppioJVM contains a straightforward JavaScript representation of the JVM call stack. Each stack frame contains an operand stack and an array of local variables. The call stack is simply an array of these stack frame objects. To ensure that execution suspends in a timely fashion, DoppioJVM checks at each backwards jump and function call whether it should suspend.

### 6.3.2 Multithreading

DoppioJVM uses Doppio's "thread pool" to emulate multiple JVM threads. DoppioJVM checks for waiting threads at fixed context switch points, such as JVM monitor checks, atomic operations, any other form of lock-checking, and each time Doppio suspends-and-resumes execution.

### 6.3.3 Native Methods

The Java Class Library exposes JVM interfaces to a wide variety of native functionality, such as the file system, unsafe memory operations, and network connections. These methods cannot be implemented using JVM bytecodes, and are marked as "native". DoppioJVM implements these native methods directly in JavaScript to interact with Doppio's operating system services.

### 6.3.4 Class Loading

The DoppioJVM class loader uses the Doppio file system to lazily download and parse JVM class files and JAR files. Using the file system, DoppioJVM makes the entire Java Class Library available in the browser.

### 6.3.5 Exceptions

The JVM is natively aware of exceptions and exception-handling logic. However, because DoppioJVM uses Doppio to execute as finite-length events, it cannot use JavaScript's native exception mechanisms to emulate JVM exceptions. Instead, DoppioJVM emulates JVM exception handling semantics by iterating through its virtual stack representation until it finds a stack frame with an applicable exception handler, or until it empties the stack and exits with an error.

### 6.3.6 JVM Objects and Arrays

DoppioJVM maps JVM objects to JavaScript objects, where each object contains a reference to its class and a dictionary that contains all of its fields keyed on their names. JVM arrays are a special type of JVM object; these are mapped to a JavaScript object that contains an array of values and a reference to the special array class that the JVM constructs according to the array's component type. DoppioJVM takes full advantage of the JavaScript garbage collector, which automatically collects JVM objects when they fall out of scope.

## 6.4 Evaluation

### 6.4.1 Case Study 1: DoppioJVM

I evaluate DoppioJVM's completeness and performance on a set of real and unmodified complex JVM programs across a wide variety of browsers. I compare DoppioJVM's performance to Oracle's HotSpot JVM interpreter provided with OpenJDK.

The benchmarks and their respective workloads are as follows: `javap` (4KLOC) is the Java disassembler from OpenJDK 6, which I run on the 491 compiled class files of `javac`. `javac` (44KLOC) is the Java compiler from OpenJDK 6, which I run on the 19 source

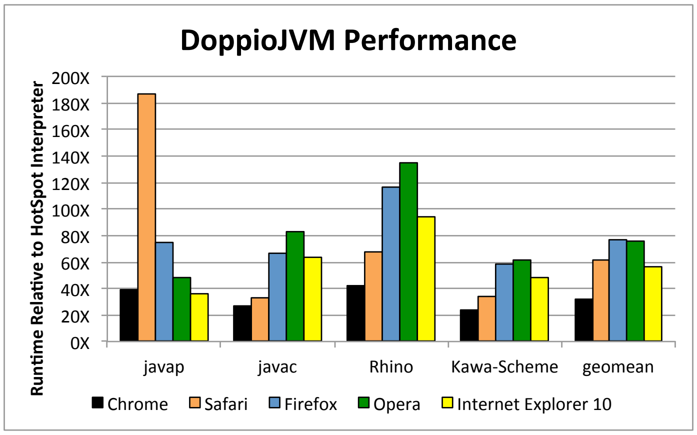


Figure 6.2: DoppioJVM's performance on the benchmark applications relative to the HotSpot JVM interpreter bundled with Java 6. DoppioJVM runs between 24× and 42× slower (geometric mean: 32×) than the HotSpot interpreter in Google Chrome. Note that `javap`'s poor performance in Safari is due to a browser bug.

files of `javap`. `Rhino` 1.7 (57KLOC) is an implementation of the JavaScript language on the JVM, which I run on the `recursive` and `binary-trees` programs from the SunSpider 0.9.1 benchmark suite. `Kawa-Scheme` 1.13 (121KLOC) is an implementation of the Scheme language on the JVM, which I evaluate on the `nqueens` algorithm with input 8.

The benchmark computer is a Mac Mini running OS X 10.8.4 with a 4-core 2GHz Intel Core i7 processor and 8GB of 1333 MHz DDR3 RAM. I evaluate DoppioJVM in Chrome 28.0, Firefox 22.0, Safari 6.0.5, Opera 12.16, and Internet Explorer 10, with Internet Explorer 10 running in a Windows 8 virtual machine using the Parallels 8 software.

DoppioJVM is able to successfully execute all of these applications to completion; I did not need to make any modifications to these applications. Figure 6.2 presents execution times across various browsers versus Oracle's HotSpot interpreter. DoppioJVM achieves its highest performance on Chrome: compared to the HotSpot interpreter, DoppioJVM runs between 24× and 42× slower (geometric mean: 32×). This performance degradation is explained by two facts: first, DoppioJVM is largely untuned; second, it pays the price for executing on top of JavaScript and inside the browser. By contrast, the HotSpot interpreter is a highly tuned native executable.

While running the `javap` benchmark, I discovered a bug in Safari that causes significant performance degradation. I have reported this issue to Apple, and it has since been fixed.[1]

### 6.4.2 Case Study 2: Doppio and C++

To further demonstrate Doppio's utility and generality, I combined Doppio with Emscripten, extending its ability to port C++ applications to the browser. As a case study, I used it to to run the C++ game *Me and My Shadow* in the browser. The Emscripten developers previously ported the core of this game to the web, but the port was incomplete: because Emscripten does not support synchronous dynamic file loading and does not back files to a persistent storage mechanism, the Emscripten demo needs to load all of the games assets into memory prior to execution and does not support game saving.

I modified Emscripten to use the Doppio file system, which is able to download the static game assets synchronously as the game requires them, and back the game's configuration folder to `localStorage`. I did not need to modify the game in order to do this; I took the same source code that the Emscripten developers used to make their demo, compiled it with an augmented version of Emscripten, and configured the Doppio file system to mount the game's resources and the browser's persistent storage at appropriate folders in the file system hierarchy. The resulting demo does not preload any files, and is able to write to the file system to save game progress and settings.

## 6.5 Conclusion

While web browsers have become ubiquitous and so are an attractive target for application developers, they support just one programming language—JavaScript—and offer an idiosyncratic execution environment that lacks many of the features that most programs require, including file systems, blocking I/O, and multiple threads. They also are incredibly diverse, further complicating the task of programming web-based applications.

This chapter presented Doppio, a runtime system for the browser that breaks the browser language barrier. Doppio addresses the challenges needed to execute programs

[1] See `https://bugs.webkit.org/show_bug.cgi?id=119049`

written in general-purpose languages inside the browser by providing key system services and runtime support that abstracts away the many differences across browsers. Using Doppio, I built DoppioJVM, a proof-of-concept complete implementation of a Java Virtual Machine in JavaScript. DoppioJVM makes it possible for the first time to run unmodified, off-the-shelf applications written in a conventional programming language directly inside the browser. DoppioJVM is already deployed as the compilation and execution engine for the educational website `CodeMoo.com`, which teaches students how to program in Java [126]. I further demonstrate Doppio's utility by combining it with Emscripten, extending its ability to port C++ applications to the browser. Tens of millions of visitors to the Internet Archive have used Doppio and Emscripten to run historical software, such as The Oregon Trail and Windows 3.1, directly in their browsers [122]. Doppio is available for download at `http://www.doppiojvm.org/`.

# CHAPTER 7

# BROWSIX: BRIDGING THE GAP BETWEEN UNIX AND THE BROWSER

The previous chapter described DOPPIO, a POSIX-like runtime system for single process applications. While DOPPIO makes it significantly easier to re-use code written in conventional languages in the browser, it can only run a single process at a time and provides limited support for sockets and synchronous I/O at the JavaScript level.

To overcome these limitations, I introduce BROWSIX, a framework that brings Unix abstractions to the browser through a shared kernel and common system-call conventions, bridging the gap between conventional operating systems and the browser [111]. BROWSIX consists of two core components: (1) a JavaScript-only operating system that exposes a wide array of OS services that applications expect (including pipes, concurrent processes, signals, sockets, and a shared file system); and (2) extended JavaScript runtimes for C, C++, Go, and Node.js that let unmodified programs written in these languages and compiled to JavaScript run directly in the browser.

## 7.1 Browsix OS Support

The core of BROWSIX's OS support is a kernel that controls access to shared Unix services. Unix services, including the shared file system, pipes, sockets, and task structures, live inside the kernel, which runs as a JavaScript library in the main browser thread. Processes run separately and in parallel inside Web Workers, and access BROWSIX kernel services through a system call abstraction.

### 7.1.1 Kernel

The kernel lives in the main JavaScript context alongside the web application and acts as the intermediary between processes and loosely coupled Unix subsystems. Processes issue

| **Class** | **System calls** |
|---|---|
| Process Management | `fork`, `spawn`, `pipe2`, `wait4`, `exit` |
| Process Metadata | `chdir`, `getcwd`, `getpid` |
| Sockets | `socket`, `bind`, `getsockname` `listen`, `accept`, `connect` |
| Directory IO | `readdir`, `getdents`, `rmdir`, `mkdir` |
| File IO | `open`, `close`, `unlink`, `llseek`, `pread`, `pwrite` |
| File Metadata | `access`, `fstat`, `lstat`, `stat`, `readlink`, `utimes` |

Table 7.1: A representative list of the system calls implemented by the Browsix kernel. `fork` is only supported for C and C++ programs.

system calls to the kernel to access shared resources, the kernel routes these requests to the appropriate subsystem, and the subsystems relay responses back to the relevant processes. The kernel also dispatches signals to processes. Table 7.1 presents a partial list of the system calls that the kernel currently supports.

### 7.1.2 System Calls

The Browsix kernel supports two types of system calls: asynchronous and synchronous. Asynchronous system calls work in all modern browsers, but impose a high performance penalty on C and C++ programs. Synchronous system calls enable higher performance for C and C++ programs, but use newly proposed browser features that are not yet available in some browsers.

**Asynchronous System Calls:** Browsix implements asynchronous system calls in a continuation-passing style (CPS). A process initiates a system call by sending a message to the kernel with a process-specific unique ID, the system call number, and arguments. The process is then required to suspend its execution, à la Doppio. When the kernel sends a response, the Web Worker process executes the continuation (or callback) with response values, resuming the process's execution.

**Synchronous System Calls:** Synchronous system calls work by sharing a view of a process's address space between the kernel and the process, similar to a traditional operating system kernel like Linux. Synchronous system calls reduce copying overhead and do not require processes to suspend their execution. This feature uses the experimental SharedArrayBuffer interface that will eventually become available in all browsers.

A process invokes a synchronous system call by sending a message as in the asynchronous case, but with arguments limited to integers and integer offsets (representing pointers) into the shared memory array. Then, the process performs a blocking wait on a specific offset into the SharedArrayBuffer and is awakened when the system call has completed or a signal is received.

### 7.1.3 Processes

Browsix relies on *Web Workers* as the foundation for emulating Unix processes. Each Browsix process has an associated task structure in the kernel that contains its process ID, parent's process ID, Web Worker object, current working directory, and map of open file descriptors. Processes can share state via the file system, send signals to one another, spawn sub-processes to perform tasks in parallel, and connect processes together using pipes. Below, I describe how Browsix maps familiar OS interfaces onto Web Workers.

**`spawn`:** `spawn` lets a process construct a new process from a specified executable on the file system. In Browsix, executables are JavaScript files or files beginning with a shebang line. When a process invokes `spawn`, Browsix creates a new task structure with the specified resources and working directory, and creates a new Web Worker that runs the target executable or interpreter.

**`fork`:** The `fork` system call creates a new process containing a copy of the current address space and call stack. Fork returns twice – first with a value of zero in the new process, and with the PID of the new process in the original. Web Workers do not expose a cloning API, and JavaScript lacks the reflection primitives required to serialize a context's entire state into a snapshot. Thus, Browsix only supports `fork` when a language runtime is able to completely enumerate and serialize its own state.

**`wait4`:** The `wait4` system call reaps child processes that have finished executing. It returns immediately if the specified child has already exited, or the `WNOHANG` option is specified. Waiting requires that the kernel not immediately free task structures, and required us to implement the zombie task state for children that have not yet been waited upon. The C library used by Emscripten, musl, uses the wait4 system call to implement the C library functions wait, wait3, and waitpid.

**exit:** Language runtimes with Browsix-support are required to explicitly issue an exit system call when they are done executing, as the containing Web Worker context has no way to know that the process has finished. This is due to the event-based nature of JavaScript environments – even if there are no pending events in the Worker's queue, the main JavaScript context could, from the perspective of the browser, send the Worker a message at any time.

### 7.1.4 Pipes

Browsix pipes are implemented as in-memory buffers with wait queues. If there is no data to be read when a process issues a read system call, Browsix enqueues the callback encapsulating the system call response which it invokes when data is written to the pipe. Similarly, if there is not enough free space in a pipe's internal buffer for a write request, Browsix waits until enough space frees up in the buffer from read requests to complete the write.

### 7.1.5 Sockets

Browsix implements a subset of the BSD/POSIX socket API, with support for SOCK_STREAM (TCP) sockets for communicating between Browsix processes. These sockets enable servers that bind, listen and then accept new connections on a socket, along with clients that connect to a socket server, with both client and server reading and writing from the connected file descriptor. Sockets are sequenced, reliable, bi-directional streams.

### 7.1.6 Shared File System

Browsix uses and extends Doppio's file system to support multiple processes. Browsix's file system adds locking operations to the filesystem to prevent operations from different processes from interleaving, which may cause the filesystem to enter a bad state. In addition, child processes inherit open file descriptors from their parents, and processes may end without closing file descriptors. Browsix uses reference counting to appropriately close open file descriptors.

## 7.2 Browsix Runtime Support

Applications invoke Browsix system calls indirectly through their runtime systems. This section describes the runtime support added to GopherJS, Emscripten, and Node.js along with the APIs exposed to web applications so they can execute programs in Browsix.

### 7.2.1 Browser Environment Extensions

Web applications run alongside the Browsix kernel in the main browser context, and have access to Browsix features through several global APIs. Browsix exposes new APIs for process creation, file access, and socket notifications, and an XMLHttpRequest-like interface to send HTTP requests to Browsix processes. Using these interfaces, developers can easily integrate Browsix processes into their web applications.

### 7.2.2 Runtime-specific Integration

For many programming languages, existing language runtimes targeted for the browser must bridge the impedance mismatch between synchronous APIs present on Unix-like systems and the asynchronous world of the browser. DoppioJVM uses Doppio's execution environment to bridge this gap in a general manner, while compile-to-JavaScript systems like Emscripten and GopherJS employ different approaches. Since Browsix supports both synchronous and asynchronous system calls, language runtimes can choose the system call convention most appropriate for their implementation.

This section describes the runtime support added to language runtimes for Go, C/C++, and Node.js. Extending Browsix support to additional language runtimes remains as future work.

**Go:** Supporting Go involves extending the existing GopherJS compiler and runtime to support issuing and waiting for system calls under Browsix. In particular, the modifications change the existing `syscall.RawSyscall` function to invoke Browsix system calls.

**C and C++:** Supporting C/C++ involves modifying Emscripten, a C and C++ compiler that targets JavaScript, to implement stubbed out system calls in its runtime. Browsix-enhanced Emscripten supports two modes - synchronous system calls and asynchronous system calls, one of which is selected at compile time. Asynchronous system calls require use of Emscripten's interpreter mode (named the "Emterpreter") to save and restore the C stack à la Doppio. With synchronous system calls, Emscripten can compile the entire application to `asm.js`, an extremely optimizable subset of JavaScript.

**Node.js:** Node.js (a.k.a. "Node") is a platform for building servers and command line tools with JavaScript, implemented in C, C++ and JavaScript on top of the V8 JavaScript engine. Node.js APIs are JavaScript modules that are loaded into the current JavaScript context. These high-level APIs are implemented in platform-agnostic JavaScript and call into lower-level C++ bindings, which in turn invoke operating system interfaces like filesystem IO, TCP sockets, and child process management. Browsix replaces these C++ bindings with pure JavaScript replacements that invoke Browsix system calls.

## 7.3 Evaluation

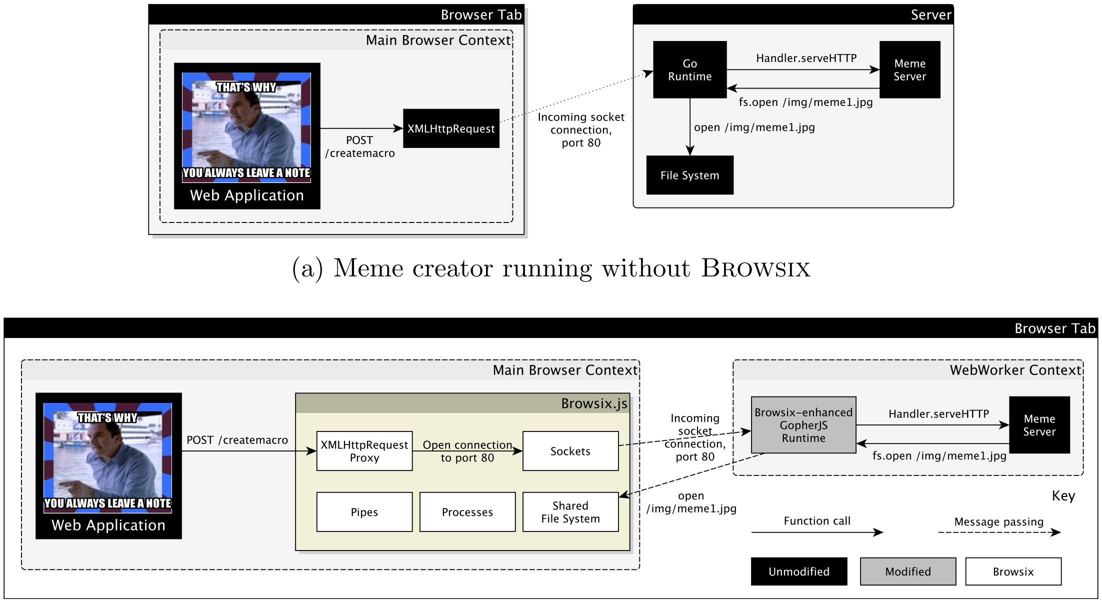
The evaluation answers the following questions: (1) Does bringing Unix abstractions into the browser enable compelling use cases? (2) Is the performance impact of running programs under Browsix acceptable?

### 7.3.1 Case Studies

This section evaluates the applicability and advantages of bringing Unix abstractions into the browser with three case studies: a LaTeX editor, a meme generator with server that can run in-browser or in the cloud, and a Unix terminal backed by `dash`, a widely-used POSIX shell.

#### 7.3.1.1 LaTeX Editor

The editor presents a split-screen view to the user, with the document's LaTeX source on the left, and generated PDF preview on the right. The editor's UI is a standard web application, and represents the only new code. When the user clicks on the "Build PDF" button, the editor uses Browsix to invoke GNU Make in a Browsix process, which rebuilds the PDF. Make then runs `pdflatex` and `bibtex`, depending on whether the user has updated the references file. Once all steps have completed, the editor reads the PDF from Browsix's file system and displays it to the user.



(a) Meme creator running without Browsix

(b) Meme creator running with Browsix

Figure 7.1: System diagram of the meme generator application with and without Browsix, demonstrating how the client and server interact with one another. With Browsix, the server runs in the browser without code modifications.

#### 7.3.1.2 Meme Generator

The meme generator lets users create *memes* consisting of images with (nominally) humorous overlaid text. Existing services, such as `MemeGenerator.net`, perform meme generation server-side. Moving meme creation into the browser would reduce server load and reduce latency when the network is overloaded or unreliable, but doing so would normally present a significant engineering challenge. The meme generation server uses sockets to communicate with the browser over HTTP and reads meme templates from the file system. Before Browsix, the client and server code would need to be re-architected and rewritten to run together in the browser.

To demonstrate Browsix's ability to quickly port code from the server to the web, the meme creator is implemented as a traditional client/server web application; Figure 7.1a contains a system diagram. The client is implemented in HTML5 and JavaScript, and the server is written in Go. The server reads base images and font files from the filesystem, and uses off-the-shelf third-party Go libraries for image manipulation and font rendering to produce memes [39]. The server also uses Go's built-in `http` module to run its web server. This server is stateless, following best practices [80]; porting a stateful server would naturally require more care.

To port the server code to Browsix, I recompile it in GopherJS with Browsix runtime extensions. I add a policy to the web application the dynamically routes meme generation requests to a server running in Browsix when running on a desktop or to the cloud when not. Figure 7.1b displays a system diagram of the modified meme generator.

#### 7.3.1.3 The Browsix Terminal

To make it easy for developers to interact with and test programs in Browsix, I implement an in-browser Unix terminal that exposes a POSIX shell. The terminal uses the Debian Almquist shell (dash), the default shell of Debian and Ubuntu. I compile `dash` to JavaScript using Browsix-enhanced Emscripten, and run it in a Browsix process.

Since the Browsix terminal uses a standard shell, developers can use it to run existing and new shell scripts in Browsix. Developers can pipe programs together (e.g. `cat file.txt | grep apple > apples.txt`), execute programs in a subshell in the background with `&`, run shell scripts, and change environment variables. Developers can also execute Go, C/C++, and Node.js programs from the shell as expected.

### 7.3.2 Performance

I evaluate the performance overhead of Browsix on two of the case studies. All experiments were performed on a late-2013 Macbook Pro with an Intel i7-4558U CPU and 16 GB of RAM, running Linux 4.8. Safari performance numbers are from the same machine running macOS Sierra.

**LaTeX Editor:** Running `pdflatex` under Browsix imposes an order of magnitude slowdown, as shown in Table 7.2. A native execution of `pdflatex` under Linux takes around 86 milliseconds on a single page document with a bibliography. When using synchronous system calls, the same document builds in Browsix in between 0.79 and 2.6 seconds, a slowdown of between 9× and 31×. Due to a limitation in Firefox, using a SharedArray-

| **Platform** | **Runtime** | |
|---|---|---|
| Linux | 0.086s | |
| Chrome Beta 56 | 2.0s | (24×) |
| Firefox Nightly 54 | 2.6s | (31×) |
| Firefox Nightly 54* | 0.79s | (9×) |
| Safari Tech Preview 22 | 1.8s | (21×) |

Table 7.2: Execution times for compiling a single-page document with a bibliography with `pdflatex` from TeX Live 2015. Times reported are the mean of 10 executions. (* Indicates Firefox was built with an 8-line patch enabling `asm.js` validation with the use of SharedArrayBuffers.)

Buffer disables certain `asm.js` optimizations in the JavaScript JIT.[1] The modified Firefox results show that if the rules for `asm.js` are slightly relaxed to allow SharedArrayBuffers for the heap, performance is improved by over 3×. Building `pdflatex` with asynchronous system calls and the Emterpreter for broader compatibility with today's browsers increases runtime to around 12 seconds.

**Meme Generator:** The meme generator performs two types of HTTP requests to the server: requests for a list of available background images, and requests to generate a meme. I benchmark the performance of the meme generator server running natively and running in Browsix in both Google Chrome 52 and Mozilla Firefox 48. Times reported are the mean of 100 runs following a 20-run warmup.

On average, a request for a list of background images takes 1.7 milliseconds natively, 9 ms in Chrome, and 6 ms in Firefox. A request to generate a meme in Browsix takes approximately two seconds, compared to 200 ms when running server-side. The former request is faster than a typical server request once roundtrip latencies are factored in, but the latter is slower, likely due to the lack of native 64-bit integers in the web platform. I expect performance to improve when future browsers support native access to 64-bit integers through WebAssembly.

[1] `https://bugzilla.mozilla.org/show_bug.cgi?id=1334941`

## 7.4 Conclusion

This chapter introduced BROWSIX, a framework that brings the essence of Unix to the browser. BROWSIX makes it almost trivial to build complex web applications from components written in a variety of languages without modifying any code, and promises to significantly reduce the effort required to build highly sophisticated web applications. BROWSIX is open source, and is freely available at `https://browsix.org`.

# CHAPTER 8
# RELATED WORK

This section describes related work on time-travel debugging and deterministic replay, memory leak debugging, understanding program behavior, and porting existing code to the browser.

## 8.1 Time-Travel Debugging and Deterministic Replay

Although McFly is the first time-traveling debugger for web applications, time-traveling debuggers exist in several other non-graphical settings (§8.1.1). Plain record-and-replay systems exist for GUI applications, but these are unable to support time-travel debugging at interactive speeds because they cannot checkpoint and roll back program and visual state (§8.1.2). Table 8.1 summarizes prior work that supports replaying web application executions.

### 8.1.1 Time-Travel Debugging

McFly is the first time-traveling debugger for web applications. Previous time-traveling debuggers for other settings fall into three main categories: *application-level* debuggers, *VM-level* debuggers, and *omniscient* debuggers.

**Application-level:** Tardis [13], Jardis [14], UndoDB [125], Boothe [20], and RR [104] record and replay program interactions with a well-defined interface to an external environment, but do not recreate state in the external environment during debugging. In other words, these debuggers do not recreate a GUI application's visual state. Tardis and Jardis debug .NET CLR and Node.js programs respectively, and replays interactions with native (C/C++) methods. UndoDB, RR, and Boothe debug the user space of processes, and replay interactions with the kernel and hardware. Boothe and RR use a similar optimization as McFly to recreate checkpoints during replay to amortize time-travel cost. RR can

| System | Program Replay | GUI Support | Debugs Web Apps | Step-backward Support |
|---|---|---|---|---|
| **Record and replay systems:** | | | | |
| Timelapse [22] | ✓ | ✓* | ✓ | |
| Mugshot [83] | ✓ | ✓* | ✓ | |
| Jalangi [116] | ✓ | | ✓ | |
| **Time-traveling debuggers:** | | | | |
| RR [104] | ✓ | | | ✓ |
| Jardis [14] | ✓ | | | ✓ |
| McFly | ✓ | ✓ | ✓ | ✓ |

Table 8.1: Comparison of prior time-traveling debuggers and record and replay systems. McFly is the first time-traveling debugger for web applications. No prior debugger is able to support step-backward debugger commands with GUI support at interactive speed. * indicates that the system does not correctly reproduce data races between the layout engine and JavaScript.

step forwards and backwards through a Firefox execution, but it is designed to debug the browser itself rather than web applications and imposes single-threaded browser execution at all times.

**VM-level:** VM-level hypervisors like XenTT [24], ReVirt [34, 35], ReTrace [151], and TTVM [67] can time-travel entire virtual machines, but at the expense of large program traces and slow time-travel. Reverse-step debugging, like that provided by McFly, requires time-travel at interactive speeds to be practical.

**Omniscient:** Omniscient debuggers provide time-traveling features by recording program state changes after every instruction, which produces large program traces and imposes high overhead during execution. Examples of omniscient debuggers include Chronon [28], TOD [110], ODB [71], and Tralfamadore [69].

### 8.1.2 Deterministic Replay

Pure deterministic replay systems can record and replay an application's execution, but do not support periodic checkpoints or reverse debugging. As a result, these systems are unable to support backwards stepping operations at interactive speeds. I center the discussion on three different runtime environments.

**Browser:** Mugshot [83] and Timelapse [22] deterministically replay web application executions by recording and replaying the event schedule and I/O operations; I compare these systems to McFly in Table 8.1. To accomplish this goal, Mugshot uses program rewriting and JavaScript reflection while Timelapse modifies the WebKit layout engine. While McFly also modifies the layout engine, Timelapse's "hypervisor-like record/replay strategy relies on the layered architecture of WebKit" and is not portable to other browsers; in contrast, McFly's architecture builds on web standards that are supported across all major browsers. Neither system is able to provide step-backward debugger commands at interactive speeds because they are unable to capture application checkpoints. Furthermore, Mugshot and Timelapse do not support layout engine operations that mutate visual state in parallel with JavaScript execution, such as CSS animations, which can cause divergent application replays.

Jalangi [116] supports selectively recording and replaying a subset of a program's code in support of dynamic analyses. On the user-selected subset of code, Jalangi logs and replays interactions with the browser's native functions with considerable overhead (26X during recording and 30x during replay), and does not support visual state during replay.

**Android:** The Android runtime environment is similar to the browser environment in that applications are event-driven and use a single thread to update the GUI. Valera [58] and ReRan [40] interpose on the interface between Android applications and the Android platform to capture nondeterministic event schedules and I/O operations.

**JVM:** JVM applications communicate with the environment and internal JVM components via native methods. Existing record-and-replay systems for the JVM treat state below the native methods, such as visual state, as external to replay. DejaVu assumes all native methods are deterministic, preventing applications from using nondeterministic APIs [25]. ORDER records and replays select nondeterministic APIs, preventing developers from inspecting or observing JVM-external state, like the GUI, during replay [152].

## 8.2 Memory Leak Debugging

BLeak is the first system for automatically debugging memory leaks in web applications. Prior techniques in this space are inapplicable or ineffective for memory leaks found on the web.

### 8.2.1 Web Application Memory Leak Detectors

BLeak automatically debugs memory leaks in modern web applications; past work in this space is ineffective, out of date, or not sufficiently general. LeakSpot locates JavaScript allocation and reference sites that produce and retain increasing numbers of objects over time, and uses staleness as a heuristic to refine its output [115]. On real web applications, LeakSpot typically reports over 50 different allocation and reference sites that developers must manually inspect to identify and diagnose memory leaks. AjaxScope dynamically detects leaks due to a bug in web browsers that has now been fixed [66]. JSWhiz statically analyzes code written with Google Closure type annotations to detect specific leak patterns [108].

### 8.2.2 Web Application Memory Debugging

Some tools help web developers debug memory usage and present diagnostic information that the developer must manually interpret to locate leaks (Section 2.2 describes Google Chrome's Development Tools). MemInsight summarizes and displays information about the JavaScript heap, including per-object-type staleness information, the allocation site of individual objects, and retaining paths in the heap [61]. Unlike BLeak, these tools do not directly identify memory as leaking or identify the code responsible for leaks.

### 8.2.3 Growth-based Memory Leak Detection

LeakBot looks for patterns in the heap graphs of Java applications to find memory leaks [91]. LeakBot assumes that leak roots own all of their leaking objects, but leaked objects in web applications frequently have multiple owners. BLeak does not rely on specific patterns, and uses round trips to the same visual state to identify leaking objects. Cork uses static type information available in the JVM to locate types that appear to be

| System | No Browser Modifications | Event Listener Registrations | Implicit Causality | Async. Ops | DOM Support | IFrame Support |
|---|---|---|---|---|---|---|
| **Program understanding**: | | | | | | |
| JSGraph [72] | | ✓ | | † | ✓ | ✓ |
| Clematis [6] | ✓ | ✓ | | † | | |
| Domino [73] | ✓ | ✓ | | ✓ | | |
| **Race detection**: | | | | | | |
| WebRacer [106] | | ✓ | | † | ✓ | ✓ |
| EventRacer [113] | | ✓ | | † | ✓ | ✓ |
| BCause | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

Table 8.2: Feature comparison of systems that reason about the causality of JavaScript events. *Implicit Causality* refers to the ability to understand program actions that trigger events (Table 5.1) and *DOM Support* refers to the ability to track causality through HTML elements in the DOM (excluding event listener registrations). A † in *Async. Ops* indicates that that system does not support Promises.

the source of memory leaks. [63]. Cork is not applicable to dynamically typed languages like JavaScript.

### 8.2.4 Staleness-based Memory Leak Detection

SWAT (C/C++), Sleigh (JVM), and Hound (C/C++) find leaking objects using a staleness metric derived from the last time an object was accessed, and identify the call site responsible for allocating them [19, 53, 103]. Leakpoint (C/C++) also identifies the last point in the execution that referenced a leaking memory location [29]. As I show (§4.4.5), staleness is ineffective for at least 77% of the memory leaks BLeak identifies .

### 8.2.5 Hybrid Leak Detection Approaches

Xu *et al.* identify leaks stemming from Java collections using a hybrid approach that targets containers that grow in size over time and contain stale items. The vast majority of memory leaks found by BLeak would not be considered stale (§4.4.5).

## 8.3 Causal Program Understanding for Web Applications

BCause is the first system to track the causality of *all* JavaScript events in an unmodified web browser. Table 8.2 summarizes prior work that reason about the causality of JavaScript events, which provide uneven support for tracking event causality through specific browser features.

**JavaScript event causality**: JSGraph [72] modifies the Chromium web browser to track DOM modifications and JavaScript execution in order to reconstruct web attacks. Clematis [6] and Domino [73] use JavaScript reflection to track event listener registrations, callback-based asynchronous operations, and DOM event execution. Neither Clematis nor Domino support tracking causality through IFrame elements or DOM mutations, which severely limits their applicability to production web applications with advertisements. Although Clematis logs DOM modifications to aid developers with debugging, it does not use this information to track event causality. Unlike prior systems, BCause supports complex production web applications containing numerous IFrames with no browser modifications.

**JavaScript race detection**: WebRacer [106] and EventRacer [113] build a happens-before graph of JavaScript events to find data races. Neither system supports tracking the happens-before relations in Table 5.1, and both require a modified WebKit-based browser. InitRacer [1] detects specific types of data races that occur during web application initialization. InitRacer does not build a happens-before graph or reason about event causality; instead, it eagerly invokes event handlers after registration and monitors the web application for uncaught exceptions.

**Fine-grained data flow tracking**: Scout [102] logs fine-grained data flows across the JavaScript heap and the DOM, and has been used to determine dependencies among network requests in order to improve page loading. It may be possible to extract event causality from Scout's detailed logs, which capture every DOM interaction and JavaScript event. However, these logs are captured using heavyweight program instrumentation. Specifically, Scout replaces all objects transitively accessible from the global scope with Proxy objects, and logs all reads and writes to them. In contrast, BCause uses WebIDL to surgically interpose only on the browser-provided interfaces related to event causality.

## 8.4 Porting Code to the Browser

Browsix and Doppio significantly extend past efforts to bring traditional APIs and general-purpose languages to the browser; Table 8.3 provides a comparison. Doppio pro-

| **System** | **Threads** | **File system** | **Socket clients** | **Socket servers** | **Processes** | **Pipes** | **Signals** |
|---|---|---|---|---|---|---|---|
| **Environments:** | | | | | | | |
| Browsix | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Doppio | † | † | † | | | | |
| `asm.js` | | | | | | | |
| WebAssembly | | | | | | | |
| **Language runtimes:** | | | | | | | |
| Emscripten (C/C++) | | † | † | † | | | |
| GopherJS (Go) | † | | | | | | |
| Browsix + Emscripten | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Browsix + GopherJS | † | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

Table 8.3: Feature comparison of JavaScript execution environments and language runtimes for programs compiled to JavaScript. † indicates that the feature is only accessible by a single running process. Browsix provides multi-process support for all of its features. Both `asm.js` and WebAssembly are pure computational environments and as such don't provide any of the OS features listed.

vides single-process POSIX abstractions, while Browsix provides Unix abstractions and builds on Doppio's file system to support multiple processes.

There are many projects that compile specific programming languages to JavaScript, but provide limited or no emulation of operating system services. To name a few, Emscripten [153] compiles LLVM bytecode to JavaScript, GopherJS compiles Go to JavaScript [101], and GWT statically compiles Java to JavaScript [51]. Doppio and Browsix are designed to provide these types of projects with the POSIX and Unix abstractions existing code expects.

Xax is a browser plugin model designed to ease porting legacy code to the web [33]. It shares a similar focus on OS independence and legacy support along with an asynchronous system-call ABI, but runs native code in a hardware-isolated picoprocess. The Xax model requires developers to compile and host binaries for all architectures an end user might use, while Browsix and Doppio leverage the cross-platform nature of JavaScript and WebAssembly to enable a compile-once, run-everywhere development workflow.

The Illinois Browser Operating System (IBOS) makes key browser APIs like HTTP requests and cookie storage OS primitives, removing library and OS code not necessary for those APIs from the trusted computing base [121]. Embassies refactors the web browser to separate the client execution interface from the developer programming interface [57]. Embassies models the client as a pico-datacenter, running each tenant (web page) in a separate VM. In addition to providing improved isolation and security for traditional web pages, Embassies enables safely executing arbitrary native code on the client in addition to a traditional WebKit HTML stack. Browsix and Doppio provide OS abstractions on top of browser APIs, and could be run in IBOS or Embassies for improved isolation and security.

# CHAPTER 9
# CONCLUSION

Although the browser is arguably the most widespread application runtime today, it remains challenging to develop web applications. Web applications must be written in JavaScript, an extraordinarily dynamic language that is designed for event-driven concurrency. JavaScript execution is tightly interwoven with the browser's GUI interface, the DOM, which obscures control flow among JavaScript events. Due to dynamism, concurrency, and the DOM, web applications exhibit complicated behavior that is difficult to debug, optimize, and understand. Developers are also unable to directly translate existing code written in C, C++, and Java into JavaScript because the browser provides an incompatible runtime environment that lacks common operating system features. Existing web development tools use techniques intended for conventional runtime environments, and provide limited assistance to developers facing these issues.

This dissertation introduces a complete set of development tools with full support for the browser environment. McFly lets developers step forwards *and backwards* through a program's execution, enabling them to trace a bug's symptom back to its root cause. BLeak automatically guides developers to code that is responsible for significant memory leaks. BCause creates a causal graph of a web application's events and forms a solid foundation for next generation browser development tools that can reason about asynchronous behavior. Doppio and Browsix enable developers to re-use well tested code written in conventional languages in the browser by bridging the gap between POSIX and Unix, respectively, and the browser. These tools make it easier for developers to write, debug, optimize, and understand their web applications.

# APPENDIX A

# LEAKS FOUND BY BLEAK

In the next few pages, I document all 59 memory leaks found by BLeak in a separate table per evaluation application. Each memory leak corresponds to a specific source code location that causes unbounded growth; in some cases, multiple memory leaks grow the same leak root or a single memory leak grows multiple leak roots. For each bug, I report the leak root, the type of the leak root, the library responsible for the unbounded growth (Culprit), whether or not the memory leak was previously known (New), if the leaked objects would be considered stale under the assumptions discussed in Section 4.4.5 (Stale), a link to the bug report (Bug), and whether or not the bug has been fixed. A ✉ in the "Bug" column indicates that I reported the bug to the culprit in an email, since the problematic code is not open source. A † in the "Fixed" column indicates that a fix is currently under code review, whereas ✓ indicates that a fix has already been merged into the codebase. A † in the "New" column indicates that the memory leak was unknown to the application developers, whereas a ✓ indicates that the memory leak was unknown to the developers of the culprit library/application.

| # | Leak Root | Type | Culprit | New | Stale | Bug | Fixed |
|---|---|---|---|---|---|---|---|
| 1 | `document.body.childNodes` | DOM | loadCSS [37] | ✓ | | [70] | † |
| 2 | ‘blur’ listeners on `window` | EL | Google Maps [48] | † | | [43] | |
| 3 | ‘blur’ listeners on `window` | EL | Airbnb | ✓ | | ✉ | |
| 4 | ‘resize’ listeners on `window` | EL | Google Maps | † | | [43] | |
| 5 | ‘click’ listeners on `document` | EL | Google Maps | † | | [43] | |
| 6 | ‘scroll’ listeners on `window` | EL | Google Maps | † | | [43] | |
| 7 | ‘scroll’ listeners on `window` | EL | Airbnb | ✓ | | ✉ | |
| 8 | ‘keydown’ listeners on `document` | EL | Google Maps | † | | [43] | |
| 9 | ‘keypress’ listeners on `document` | EL | Google Maps | † | | [43] | |
| 10 | `document.__e3_[‘keydown’]` | Obj. | Google Maps | † | | [43] | |
| 11 | ‘keyup’ listeners on `document` | EL | Google Maps | † | | [43] | |
| 12 | `__e3_[‘resize’]` | Obj. | Google Maps | † | | [43] | |
| 13 | `document.__e3_[‘keyup’]` | Obj. | Google Maps | † | | [43] | |
| 14 | `document.__e3_[‘click’]` | Obj. | Google Maps | † | | [43] | |
| 15 | `__e3_[‘blur’]` | Obj. | Google Maps | † | | [43] | |
| 16 | `document.__e3_[‘keypress’]` | Obj. | Google Maps | † | | [43] | |
| 17 | ‘MSFullscreenChange’ listeners on `document` | EL | Google Maps | † | | [43] | |
| 18 | ‘fullscreenchange’ listeners on `document` | EL | Google Maps | † | | [43] | |
| 19 | ‘mozfullscreenchange’ listeners on `document` | EL | Google Maps | † | | [43] | |
| 20 | ‘webkitfullscreenchange’ listeners on `document` | EL | Google Maps | † | | [43] | |
| 21 | `document.__e3_ [‘fullscreenchange’]` | Obj. | Google Maps | † | | [43] | |
| 22 | `document.__e3_ [‘mozfullscreenchange’]` | Obj. | Google Maps | † | | [43] | |
| 23 | `document.__e3_ [‘webkitfullscreenchange’]` | Obj. | Google Maps | † | | [43] | |
| 24 | `document.__e3_ [‘MSFullscreenChange’]` | Obj. | Google Maps | † | | [43] | |
| 25 | `document.head.childNodes` | DOM | GTM [49] | ✓ | | [135] | |
| 26 | `_xdc_` | Obj. | Google Maps | ✓ | ✓ | [141] | ✓ |
| 27 | ‘focus’ listeners on `window` | EL | Airbnb | ✓ | | ✉ | |
| 28 | `ga.h.t0.b.data.keys` | Array | G. Analytics [45] | ✓ | ✓ | [140] | |
| 29 | `document.body. childNodes[126].childNodes` | DOM | Criteo One-Tag [32] | ✓ | | ✉ | † |
| 30 | `e.extraData` in closure of `criteo_q.push` | Array | Criteo OneTag | ✓ | ✓ | ✉ | † |
| 31 | `A` in closure of `__inner__` property on the second ‘popstate’ listener of `window` | Array | Airbnb | ✓ | ✓ | ✉ | |
| 32 | `n[‘5v9T’].exports._events ['header:search']` within closure of `webpackJsonp` | Array | Airbnb | ✓ | | ✉ | |

Table A.1: Memory leaks in Airbnb found by BLeak

| # | Leak Root | Type | Culprit | New | Stale | Bug | Fixed |
|---|---|---|---|---|---|---|---|
| 33 | `jQuery22305631933622062061 .events.resize` | Array | Piwik | ✓ |  | [130] | ✓ |
| 34 | `jQuery22305631933622062061 .events.resize` | Array | Piwik | ✓ |  | [130] | ✓ |
| 35 | `jQuery22305631933622062061 .events.resize` | Array | Piwik | ✓ |  | [131] | ✓ |
| 36 | `bb` in closure of `Raphael` | Obj. | Raphael.js [12] | ✓ | ✓ | [129] | † |
| 37 | `body.jQuery22305631933622062061 .events.mouseup` in closure of `$widgetContent.__proto__.mwheelIntent` | Array | Piwik | ✓ |  | [130] | ✓ |
| 38 | `document.body.childNodes` | DOM | Piwik | ✓ |  | [131] | ✓ |
| 39 | `allRequests` in closure of `a.piwikApi.withTokenInUrl` in closure of `Ea.jQuery22305631933622062062.$injector .invoke` in closure of `jQuery` | Piwik | Array | ✓ | ✓ | [139] | ✓ |
| 40 | `$widgetContent['0'] .jQuery22305631933622062062.$scope .$$listeners.$destroy` | Array | Piwik | ✓ | ✓ | [134] |  |
| 41 | `jQuery22305631933622062061 .events.click` | Array | Materialize [79] | ✓ |  | [137] | ✓ |
| 42 | `piwik.UI.UIControl._controls` | Array | Piwik | ✓ | ✓ | [132] | ✓ |
| 43 | Property `jQuery22305631933622062061 .events.click` on all `div` children of `#columnPreview` | Array | Piwik | ✓ |  | [133] | ✓ |

Table A.2: Memory leaks in Piwik found by BLeak

| # | Leak Root | Type | Culprit | New | Stale | Bug | Fixed |
|---|---|---|---|---|---|---|---|
| 44 | `angular.element.cache[3] .events['resize']` | Array | Ment.io [30] | † | | [148] | † |
| 45 | `angular.element.cache[2] .events['click']` | Array | Ment.io | † | | [148] | † |
| 46 | `angular.element.cache[2] .events['paste']` | Array | Ment.io | † | | [148] | † |
| 47 | `angular.element.cache[2] .events['keypress']` | Array | Ment.io | † | | [148] | † |
| 48 | `angular.element.cache[2] .events['keydown']` | Array | Ment.io | † | | [148] | † |
| 49 | `Loomio.records.discussions.collection .DynamicViews` | Array | Loomio | ✓ | ✓ | [138] | ✓ |
| 50 | `angular.element.cache[4].data.$scope .$parent.$$listeners .$translateChangeSuccess` | Array | AngularJS (1.x) [46] | ✓ | ✓ | [142] | ✓ |
| 51 | `Loomio.records.stanceChoices .collection.DynamicViews` | Array | Loomio | ✓ | ✓ | [138] | ✓ |
| 52 | `Loomio.records.versions.collection .DynamicViews` | Array | Loomio | ✓ | ✓ | [138] | ✓ |

Table A.3: Memory leaks in Loomio found by BLeak

| # | Leak Root | Type | Culprit | New | Stale | Bug | Fixed |
|---|---|---|---|---|---|---|---|
| 53 | `list` in closure of `tuples[0][3].add` in closure of `$.ready.then`, and `list` in closure of `tuples[2][3].add` in closure of `$.ready.then` | Array | jQuery [123] | † | ✓ | [18] | † |
| 54 | `EventLog.eventbindings` | Array | Mailpile | ✓ | | [127] | ✓ |
| 55 | `document.body.childNodes[3] .childNodes[3].childNodes` | DOM | Mailpile | ✓ | | [128] | ✓ |

Table A.4: Memory leaks in Mailpile found by BLeak

| # | Leak Root | Type | Culprit | New | Stale | Bug | Fixed |
|---|---|---|---|---|---|---|---|
| 56 | 'mouseover' listeners on `cm.display.wrapper` | EL | Firefox debugger | ✓ | | [136] | ✓ |
| 57 | 'mouseup' listeners on `cm.display.wrapper` | EL | Firefox debugger | ✓ | | [136] | ✓ |
| 58 | 'mousedown' listeners on `cm.display.wrapper` | EL | Firefox debugger | ✓ | | [136] | ✓ |
| 59 | `cm._handlers.scroll` | Array | Firefox debugger | ✓ | | [136] | ✓ |

Table A.5: Memory leaks in the Firefox debugger found by BLeak

# APPENDIX B

# BLEAK EVALUATION APPLICATION LOOPS

This section lists the code for all of the loops used in the evaluation (§7.3). These scripts are the only input BLeak needs to automatically locate, rank, and debug the memory leaks from the evaluation. Note that the line counts reported in Figure 4.1 ignore comment lines.

```
exports.loop = [{
  check: function() {
    const buttons = document.getElementsByTagName('button');
    return document.getElementsByTagName('a')[4].getAttribute('aria-selected') ===
     ↪  'true';
  },
  next: function() {
    document.getElementsByTagName('a')[5].click();
  }
}, {
  check: function() {
    const buttons = document.getElementsByTagName('button');
    return document.getElementsByTagName('a')[5].getAttribute('aria-selected') ===
     ↪  'true' && buttons.length > 11 && buttons[11].innerText.trim() === "Room type"
  },
  next: function() {
    document.getElementsByTagName('a')[4].click();
  }
}];
```

Table B.1: Airbnb's loop.

```
exports.login = [
  {
    check: function() {
      const input = document.getElementsByTagName('input');
      const username = input[0];
      const password = input[2];
      const submit = document.getElementsByClassName('submit')[0];
      return !!(username && password && submit);
    },
    next: function() {
      const input = document.getElementsByTagName('input');
      const username = input[0];
      const password = input[2];
      const submit = document.getElementsByClassName('submit')[0];
      username.value = "bleak";
      password.value = "bleakpldi";
      submit.click();
    }
  }
];
exports.loop = [
  {
    check: function() {
      const svg = document.getElementsByTagName('svg');
      const canvas = document.getElementsByTagName('canvas');
      return svg.length === 1 && canvas.length === 42;
    },
    next: function() {
      document.getElementsByClassName('item')[1].click();
    }
  }
];
```

Table B.2: Piwik's loop.

```
exports.login = [{
  check: function() {
    const emailField = document⌋
     ↪   .getElementsByTagName('input')[1];
    if (emailField) {
      return emailField.getAttribute('name')
       ↪   === 'email';
    }
    return false;
  },
  next: function() {
    const emailField = document⌋
     ↪   .getElementsByTagName('input')[1];
    emailField.value = 'default@loomio.org';
    // Notify Angular code of change.
    emailField.dispatchEvent(new
     ↪   Event("change"));
    const submitBtn = document⌋
     ↪   .getElementsByTagName('button')[2];
    submitBtn.click();
  }
}, {
  check: function() {
    const pswdField = document⌋
     ↪   .getElementsByTagName('input')[1];
    const modalHeader = document⌋
     ↪   .getElementsByTagName('h2')[3];
    const submitBtn = document⌋
     ↪   .getElementsByTagName('button')[3];
    return submitBtn && pswdField &&
     ↪   pswdField.name === "password" &&
     ↪   modalHeader &&
     ↪   modalHeader.innerText === "Welcome
     ↪   back, default@loomio.org!" &&
     ↪   submitBtn.innerText === "SIGN IN";
  },
  next: function() {
    const pswdField = document⌋
     ↪   .getElementsByTagName('input')[1];
    pswdField.value = 'b0eb3a48';
    pswdField.dispatchEvent(new
     ↪   Event("change"));
    const submitBtn = document⌋
     ↪   .getElementsByTagName('button')[3];
    submitBtn.click();
  }
}];
exports.setup = [{
  check: function() {
    const tp = document⌋
     ↪   .getElementsByClassName('thread-preview');
    if (tp.length > 0) {
      const thread = tp[0];
      return thread.childNodes.length > 0 &&
       ↪   thread.childNodes[0].tagName ===
       ↪   "A" && thread.childNodes[0]⌋
       ↪   .getAttribute('href') ===
       ↪   "/d/6jZ4c8FL/how-to-use-loomio";
    }
    return false;
  },
  next: function() {
    document⌋
     ↪   .getElementsByTagName('md_icon_button')[0]⌋
     ↪   .click();
  }
}];
exports.loop = [{
  check: function() {
    const span = document⌋
     ↪   .getElementsByTagName('span')[6];
    return !!span && span.innerText === "Fun
     ↪   Group 1";
  },
  next: function() {
    document⌋
     ↪   .getElementsByTagName('span')[6]⌋
     ↪   .click();
  }
}, {
  check: function() {
    const tp = document⌋
     ↪   .getElementsByClassName('thread-preview');
    if (tp.length > 0) {
      const thread = tp[0];
      return thread.childNodes.length > 0 &&
       ↪   thread.childNodes[0].tagName ===
       ↪   "A" && thread.childNodes[0]⌋
       ↪   .getAttribute('href') ===
       ↪   "/d/6jZ4c8FL/how-to-use-loomio";
    }
    return false;
  },
  next: function() {
    document⌋
     ↪   .getElementsByClassName('thread-preview')[0]⌋
     ↪   .childNodes[0].click();
  }
}, {
  check: function() {
    const paragraphs =
     ↪   document.getElementsByTagName('p');
    const h3 = document⌋
     ↪   .getElementsByTagName('h3')[3];
    return paragraphs.length > 6 && h3 &&
     ↪   h3.innerText.indexOf("Loomio Helper
     ↪   Bot started a proposal") === 0 &&
     ↪   paragraphs[5].innerText ===
     ↪   "Welcome to Loomio, an online place
     ↪   to make decisions together.";
  },
  next: function() {
    // Opens menu w/ logout.
    document⌋
     ↪   .getElementsByTagName('md_icon_button')[0]⌋
     ↪   .click();
  }
}];
```

Table B.3: Loomio's loop.

```
const emailSubjects = [ "YO DAWG", "Icelandic Banana", "Demo ipsum",
  "CRYPTO-GRAM, January 15, 2014"];
function returnToInbox() {
  document.getElementById('sidebar-tag-b').children[0].click();
}
function itemSteps(i) {
  let lastCheck = 0;
  function inboxCheck() {
    return document.getElementsByClassName('message-subject').length === 0 &&
     ↪   document.getElementsByClassName('item-subject').length > 0;
  }
  function inboxClick() {
    document.getElementsByClassName('item-subject')[i].click()
  }
  return [{
    check: inboxCheck,
    next: inboxClick
  }, {
    check: function() {
      const ms = document.getElementsByClassName('message-subject');
      const rv = ms.length === 1 && ms[0].innerText === emailSubjects[i];
      const now = Date.now();
      // Mailpile's server fails a *lot*, requiring multiple clicks to get through.
      // This is a hack to get around what I consider to be a bug.
      if (!rv && inboxCheck() && (now - lastCheck) > 1000) {
        inboxClick();
      }
      lastCheck = now;
      return rv;
    },
    next: function() {
      lastCheck = 0;
      returnToInbox();
    }
  }];
}
exports.loop = [];
for (let i = 0; i < emailSubjects.length; i++) {
  exports.loop.push.apply(exports.loop, itemSteps(i));
}
```

Table B.4: Mailpile's loop.

```
exports.loop = [{
  check: function() {
    const nodes = document.getElementsByClassName('node');
    const sourceTabs = document.getElementsByClassName('source-tab');
    return sourceTabs.length === 0 && nodes.length > 1 && nodes[1].innerText ===
     ↪  "main.js";
  },
  next: function() {
    document.getElementsByClassName('node')[1].click();
  }
}, {
  check: function() {
    // code mirror must be open, tab must be added, etc.
    return document.getElementsByClassName('CodeMirror-line').length > 2 &&
     ↪  document.getElementsByClassName('source-tab').length === 1 &&
     ↪  document.getElementsByClassName('close-btn').length === 1;
  },
  next: function() {
    document.getElementsByClassName('close-btn')[0].click();
  }
}];
```

Table B.5: Firefox Debugger's loop.

# BIBLIOGRAPHY

[1] Adamsen, Christoffer Quist, Møller, Anders, and Tip, Frank. Practical initialization race detection for JavaScript web applications. *Proceedings of the ACM on Programming Languages 1*, OOPSLA (2017), 66:1–66:22.

[2] Adobe Corporate Communications. Flash & The Future of Interactive Content. `https://theblog.adobe.com/adobe-flash-update/`, 2017. [Online; accessed 16-July-2018].

[3] Adobe Systems Inc. Adobe Flash Player Download. `https://get.adobe.com/flashplayer/`, 2018. [Online; accessed 16-July-2018].

[4] Adya, Atul, Howell, Jon, Theimer, Marvin, Bolosky, William J., and Douceur, John R. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference, General Track* (2002), pp. 289–302.

[5] Airbnb, Inc. Vacation Rentals, Homes, Experiences, & Places - Airbnb. `http://airbnb.com/`, 2017. [Online; accessed 12-October-2017].

[6] Alimadadi, Saba, Sequeira, Sheldon, Mesbah, Ali, and Pattabiraman, Karthik. Understanding JavaScript Event-Based Interactions with Clematis. *ACM Transactions on Software Engineering and Methodology 25*, 2 (2016), 12:1–12:38.

[7] Apple. Safari Web Development Tools. `https://developer.apple.com/safari/tools/`. Accessed: 2017-04-30.

[8] Apple. JavaScriptCore - Apple Developer Documentation. `https://developer.apple.com/documentation/javascriptcore`, 2018. [Online; accessed 16-July-2018].

[9] Apple Inc. Tools - Safari for Developers. `https://developer.apple.com/safari/tools/`, 2018. [Online; accessed 10-April-2018].

[10] Apple Inc. Webkit. `https://webkit.org/`, 2018. [Online; accessed 10-April-2018].

[11] Babel. Babel - The compiler for writing next generation JavaScript. `https://babeljs.io/`, 2017. [Online; accessed 15-October-2017].

[12] Baranovskiy, Dmitry. DmitryBaranovskiy/raphael: JavaScript Vector Library. `https://github.com/DmitryBaranovskiy/raphael`, 2017. [Online; accessed 6-November-2017].

[13] Barr, Earl T., and Marron, Mark. Tardis: Affordable time-travel debugging in managed runtimes. In *Proceedings of the 2014 Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2014), pp. 67–82.

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

[15] Basques, Kayce. Fix Memory Problems. `https://developers.google.com/web/tools/chrome-devtools/memory-problems/`, 2017. [Online; accessed 2-November-2017].

[16] Basques, Kayce. Get Started with Debugging JavaScript in Chrome DevTools. `https://developers.google.com/web/tools/chrome-devtools/javascript/`, 2018. [Online; accessed 10-April-2018].

[17] Basques, Kayce. Performance Analysis Reference. `https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/reference`, 2018. [Online; accessed 25-July-2018].

[18] Bedard, Jason. Deferred: fix memory leak of promise callbacks. `https://github.com/jquery/jquery/pull/3657`, 2017. [Online; accessed 8-November-2017].

[19] Bond, Michael D., and McKinley, Kathryn S. Bell: bit-encoding online memory leak detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (2006), pp. 61–72.

[20] Boothe, Bob. Efficient algorithms for bidirectional debugging. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2000), pp. 299–310.

[21] browserbench.org. Speedometer 2.0. `https://browserbench.org/Speedometer2.0/`, 2018. [Online; accessed 25-July-2018].

[22] Burg, Brian, Bailey, Richard, Ko, Andrew J., and Ernst, Michael D. Interactive record/replay for web application debugging. In *Proceedings of the 26th Symposium on User Interface Software and Technology* (2013), pp. 473–484.

[23] Burmister, Adam. Flog.RayTracer Canvas Demo. `https://web.archive.org/web/20120218103726/http://labs.flog.co.nz:80/raytracer`, 2012. [Online; accessed 16-April-2018].

[24] Burtsev, Anton, Johnson, David, Hibler, Mike, Eide, Eric, and Regehr, John. Abstractions for Practical Virtual Machine Replay. In *Proceedings of the 12th Conference on Virtual Execution Environments* (2016), pp. 93–106.

[25] Choi, Jong-Deok, and Srinivasan, Harini. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools* (1998), pp. 48–59.

[26] Chrome Platform Status. Shared Array Buffers, Atomics and Futex APIs. `https://www.chromestatus.com/feature/4570991992766464`, 2018. [Online; accessed 16-July-2018].

[27] Chromium Contributors. Blink - The Chromium Projects. `https://www.chromium.org/blink`, 2018. [Online; accessed 10-April-2018].

[28] Chronon Systems. Chronon, a DVR for Java. `http://chrononsystems.com`, 2017. [Online; accessed 30-April-2017].

[29] Clause, James A., and Orso, Alessandro. LEAKPOINT: pinpointing the causes of memory leaks. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (2010), pp. 515–524.

[30] Collins, Jeff. jeff-collins/ment.io: Mentions and Macros for Angular. `https://github.com/jeff-collins/ment.io`, 2017. [Online; accessed 6-November-2017].

[31] Cortesi, Aldo, Hils, Maximilian, Kriechbaumer, Thomas, and contributors. mitmproxy: A free and open source interactive HTTPS proxy, 2010. [Online; accessed 15-October-2017].

[32] Criteo. Criteo OneTag Explained. `https://support.criteo.com/hc/en-us/articles/202726972-Criteo-OneTag-explained`, 2017. [Online; accessed 6-November-2017].

[33] Douceur, John R., Elson, Jeremy, Howell, Jon, and Lorch, Jacob R. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (2008), pp. 339–354.

[34] Dunlap, George W., King, Samuel T., Cinar, Sukru, Basrai, Murtaza A., and Chen, Peter M. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *Proceedings of the 5th Symposium on Operating System Design and Implementation* (2002).

[35] Dunlap, George W., Lucchetti, Dominic G., Fetterman, Michael A., and Chen, Peter M. Execution replay of multiprocessor virtual machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments* (2008), pp. 121–130.

[36] EasyList Contributors. EasyList and EasyPrivacy - Overview. `https://easylist.to/`, 2018. [Online; accessed 28-July-2018].

[37] Filament Group, Inc. filamentgroup/loadCSS: A function for loading CSS asynchronously. `https://github.com/filamentgroup/loadCSS`, 2017. [Online; accessed 6-November-2017].

[38] Firefox Developer Tools Team. debugger.html: The Firefox debugger that works anywhere. `http://firefox-dev.tools/debugger.html/`, 2017. [Online; accessed 12-October-2017].

[39] Fogleman, Michael. fogleman/gg: Go Graphics - 2D rendering in Go with a simple API. `https://github.com/fogleman/gg`, 2016. [Online; accessed 6-May-2016].

[40] Gomez, Lorenzo, Neamtiu, Iulian, Azim, Tanzirul, and Millstein, Todd D. RERAN: timing- and touch-sensitive record and replay for Android. In *Proceedings of the 35th International Conference on Software Engineering* (2013), pp. 72–81.

[41] Google. Chrome DevTools. `https://developers.google.com/web/tools/chrome-devtools/`. [Online; accessed 30-April-2017].

[42] Google. Octane v1. https://developers.google.com/octane/. [Online; accessed 30-April-2017].

[43] Google. Bug: Destroying Google Map Instance Never Frees Memory. https://issuetracker.google.com/issues/35821412, 2011. [Online; accessed 2-November-2017].

[44] Google. Saying Goodbye to Our Old Friend NPAPI. https://blog.chromium.org/2013/09/saying-goodbye-to-our-old-friend-npapi.html, 2013. [Online; accessed 16-July-2018].

[45] Google. Adding analytics.js to Your Site. https://developers.google.com/analytics/devguides/collection/analyticsjs/, 2017. [Online; accessed 6-November-2017].

[46] Google. angular/angular.js: AngularJS - HTML enhanced for web apps! https://github.com/angular/angular.js, 2017. [Online; accessed 6-November-2017].

[47] Google. Chrome DevTools Protocol Viewer. https://chromedevtools.github.io/devtools-protocol/, 2017. [Online; accessed 7-November-2017].

[48] Google. Google Maps JavaScript API. https://developers.google.com/maps/documentation/javascript/, 2017. [Online; accessed 6-November-2017].

[49] Google. Tag Management Solutions for Web and Mobile. https://www.google.com/analytics/tag-manager/, 2017. [Online; accessed 6-November-2017].

[50] Google. Chrome v8. https://developers.google.com/v8/, 2018. [Online; accessed 16-July-2018].

[51] Google Web Toolkit Community. GWT. http://www.gwtproject.org/, 2018. [Online; accessed 17-July-2018].

[52] Hara, Kentaro. State of Blink's Speed. https://docs.google.com/presentation/d/1Az-F3CamBq6hZ5QqQt-ynQEMWEhHY1VTvlRwL7b_6TU, 2017. See slide 46. [Online; accessed 2-November-2017].

[53] Hauswirth, Matthias, and Chilimbi, Trishul M. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems* (2004), pp. 156–164.

[54] Henry, Alan. How Do I Stop My Browser from Slowing to a Crawl? https://lifehacker.com/5833074/how-do-i-stop-my-browser-from-slowing-to-a-crawl, 2011. [Online; accessed 4-November-2017].

[55] Hickson, Ian. Web Storage (Second Edition). http://www.w3.org/TR/webstorage/, 2015. [Online; accessed 30-April-2017].

[56] Hors, Arnaud Le, Hégaret, Philippe Le, Wood, Lauren, Nicol, Gavin, Robie, Jonathan, Champion, Mike, and Byrne, Steve. Document Object Model (DOM) Level 3 Core Specification. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/, 2004. [Online; accessed 30-April-2017].

[57] Howell, Jon, Parno, Bryan, and Douceur, John R. Embassies: Radically refactoring the web. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013), pp. 529–546.

[58] Hu, Yongjian, Azim, Tanzirul, and Neamtiu, Iulian. Versatile yet lightweight record-and-replay for android. In *Proceedings of the 2015 Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2015), pp. 349–366.

[59] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual V3. `http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html`, 2015. [Online; accessed 30-April-2017].

[60] James Gosling and Bill Joy and Guy Steele and Gilad Bracha and Alex Buckley. The Java Language Specification. `http://docs.oracle.com/javase/specs/jls/se8/html/index.html`. [Online; accessed 30-April-2017].

[61] Jensen, Simon Holm, Sridharan, Manu, Sen, Koushik, and Chandra, Satish. MemInsight: Platform-Independent Memory Debugging for JavaScript. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015), pp. 345–356.

[62] Jin, Linghua. Angular-ColorGame. `https://github.com/linghuaj/Angular-ColorGame/tree/angular1`, 2016. [Online; accessed 30-April-2017].

[63] Jump, Maria, and McKinley, Kathryn S. Cork: dynamic memory leak detection for garbage-collected languages. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2007), pp. 31–38.

[64] Kearney, Meggin. How to Record Heap Snapshots. `https://developers.google.com/web/tools/chrome-devtools/memory-problems/heap-snapshots`, 2017. [Online; accessed 11-November-2017].

[65] Keizer, Gregg. Microsoft nixes ActiveX add-on technology in new Edge browser. `https://www.computerworld.com/article/2920892/web-browsers/microsoft-nixes-activex-add-on-technology-in-new-edge-browser.html`, 2015. [Online; accessed 16-July-2018].

[66] Kiciman, Emre, and Livshits, Benjamin. AjaxScope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (2007), pp. 17–30.

[67] King, Samuel T., Dunlap, George W., and Chen, Peter M. Debugging Operating Systems with Time-Traveling Virtual Machines. In *Proceedings of the 2005 USENIX Annual Technical Conference* (2005), pp. 1–15.

[68] Lee, Loreena, and Hundt, Robert. BloatBusters: Eliminating memory leaks in Gmail. `https://docs.google.com/presentation/d/1wUVmf78gG-ra5a0xvTfYdiLkdGaR9OhXRn0lIcEmu2s`, 2012. [Online; accessed 2-November-2017].

[69] Lefebvre, Geoffrey, Cully, Brendan, Head, Christopher, Spear, Mark, Hutchinson, Norman C., Feeley, Mike, and Warfield, Andrew. Execution mining. In *Proceedings of the 8th International Conference on Virtual Execution Environments* (2012), pp. 145–158.

[70] Lencioni, Joe. Possible memory leak when used with the same URL multiple times. `https://github.com/filamentgroup/loadCSS/issues/236`, 2017. [Online; accessed 6-November-2017].

[71] Lewis, Bill. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated Debugging* (2003).

[72] Li, Bo, Vadrevu, Phani, Lee, Kyu Hyung, and Perdisci, Roberto. JSgraph: Enabling Reconstruction of Web Attacks via Efficient Tracking of Live In-Browser JavaScript Executions. In *25th Annual Network and Distributed System Security Symposium* (2018).

[73] Li, Ding, Mickens, James, Nath, Suman, and Ravindranath, Lenin. Domino: understanding wide-area, asynchronous event causality in web applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (2015), pp. 182–188.

[74] Lo, James Teng Kin, Wohlstadter, Eric, and Mesbah, Ali. Imagen: Runtime migration of browser sessions for JavaScript web applications. In *22nd International World Wide Web Conference* (2013), pp. 815–826.

[75] Loomio Cooperative Limited. Loomio - Better decisions together. `https://www.loomio.org/`, 2017. [Online; accessed 12-October-2017].

[76] Mailpile Team. Mailpile Demo's mailpile v1.0.0rc0. `https://demo.mailpile.is/in/inbox/`, 2017. [Online; accessed 8-November-2017].

[77] Mailpile Team. Mailpile: e-mail that protects your privacy. `http://mailpile.is/`, 2017. [Online; accessed 12-October-2017].

[78] Martin, Joel. kanaka/websockify. `https://github.com/novnc/websockify`. [Online; accessed 30-April-2017].

[79] Materialize. Dogfalo/materialize: Materialize, a CSS Framework based on Material Design. `https://github.com/Dogfalo/materialize`, 2017. [Online; accessed 6-November-2017].

[80] Mauro, Tony. *Adopting Microservices at Netflix: Lessons for Architectural Design*, 2015. `https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/`.

[81] McCormack, Cameron. Web IDL Specification. `https://www.w3.org/TR/2012/CR-WebIDL-20120419/`, 2012. [Online; accessed 30-April-2017].

[82] McElhearn, Kirk. It's time for Safari to go on a memory diet. `https://www.macworld.com/article/3148256/browsers/it-s-time-for-safari-to-go-on-a-memory-diet.html`, 2016. [Online; accessed 4-November-2017].

[83] Mickens, James, Elson, Jeremy, and Howell, Jon. Mugshot: Deterministic capture and replay for Javascript applications. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation* (2010), pp. 159–174.

[84] Microsoft. PerformanceCounter Class. `https://msdn.microsoft.com/en-us/library/system.diagnostics.performancecounter(v=vs.110).aspx`. [Online; accessed 30-April-2017].

[85] Microsoft. Microsoft Edge F12 DevTools - Memory. `https://docs.microsoft.com/en-us/microsoft-edge/f12-devtools-guide/memory`, 2017. [Online; accessed 4-November-2017].

[86] Microsoft. Microsoft Edge Devtools - Performance. `https://docs.microsoft.com/en-us/microsoft-edge/devtools-guide/performance`, 2018. [Online; accessed 25-July-2018].

[87] Microsoft. Microsoft Silverlight. `https://www.microsoft.com/silverlight/`, 2018. [Online; accessed 16-July-2018].

[88] Microsoft Corporation. Microsoft ChakraCore. `https://github.com/Microsoft/ChakraCore`. [Online; accessed 30-April-2017].

[89] Microsoft Corporation. Time Travel Debugging with Node-ChakraCore (Based on McFly). `https://aka.ms/NodeTTD`. [Online; accessed 30-April-2017].

[90] Microsoft Inc. Microsoft Edge Devtools - Debugger. `https://docs.microsoft.com/en-us/microsoft-edge/devtools-guide/debugger`, 2017. [Online; accessed 10-April-2018].

[91] Mitchell, Nick, and Sevitsky, Gary. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. In *Proceedings of ECOOP 2003 - Object-Oriented Programming* (2003), pp. 351–377.

[92] Mondello, Ricky. Next Steps for Legacy Plug-ins. `https://webkit.org/blog/6589/next-steps-for-legacy-plug-ins/`, 2016. [Online; accessed 16-July-2018].

[93] Mozilla. Gecko - mdn. `https://developer.mozilla.org/en-US/docs/Mozilla/Gecko`, 2016. [Online; accessed 10-April-2018].

[94] Mozilla. Debugger - Firefox Developer Tools. `https://developer.mozilla.org/en-US/docs/Tools/Debugger`, 2017. [Online; accessed 10-April-2018].

[95] Mozilla. Firefox uses too much memory (RAM) - How to fix. `https://support.mozilla.org/en-US/kb/firefox-uses-too-much-memory-ram`, 2017. [Online; accessed 4-November-2017].

[96] Mozilla. Memory - Firefox Developer Tools. `https://developer.mozilla.org/en-US/docs/Tools/Memory`, 2017. [Online; accessed 11-November-2017].

[97] Mozilla. SensorWeb. `http://aws-sensorweb-static-site.s3-website-us-west-2.amazonaws.com/`, 2017. [Online; accessed 2-November-2017].

[98] Mozilla. Performance - Firefox Developer Tools. https://developer.mozilla.org/en-US/docs/Tools/Performance, 2018. [Online; accessed 25-July-2018].

[99] Mozilla. SpiderMonkey - MDN. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey, 2018. [Online; accessed 16-July-2018].

[100] Mozilla Development Network. Proxy - JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy, 2017. [Online; accessed 7-November-2017].

[101] Musiol, Richard. gopherjs/gopherjs: A compiler from Go to JavaScript for running Go code in a browser. https://github.com/gopherjs/gopherjs, 2016. [Online; accessed 6-May-2016].

[102] Netravali, Ravi, Goyal, Ameesh, Mickens, James, and Balakrishnan, Hari. Polaris: Faster page loads using fine-grained dependency tracking. In *13th USENIX Symposium on Networked Systems Design and Implementation* (2016), pp. 123–136.

[103] Novark, Gene, Berger, Emery D., and Zorn, Benjamin G. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2009), pp. 397–407.

[104] O'Callahan, Robert, Jones, Chris, Froyd, Nathan, Huey, Kyle, Noll, Albert, and Partush, Nimrod. Engineering record and replay for deployability: Extended technical report. *Computing Research Repository abs/1705.05937* (2017).

[105] Pecoraro, Joseph. Memory Debugging with Web Inspector. https://webkit.org/blog/6425/memory-debugging-with-web-inspector/, 2016. [Online; accessed 4-November-2017].

[106] Petrov, Boris, Vechev, Martin T., Sridharan, Manu, and Dolby, Julian. Race detection for web applications. In *Proceedings of the 2012 Conference on Programming Language Design and Implementation* (2012), pp. 251–262.

[107] Pflug, Kyle. Introducing EdgeHTML 13, our first platform update for Microsoft Edge. https://blogs.windows.com/msedgedev/2015/11/16/introducing-edgehtml-13-our-first-platform-update-for-microsoft-edge, 2015. [Online; accessed 10-April-2018].

[108] Pienaar, Jacques A., and Hundt, Robert. JSWhiz: Static analysis for JavaScript memory leaks. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization* (2013), pp. 11:1–11:11.

[109] Piwik.org. #1 Free Web & Mobile Analytics Software. https://piwik.org/, 2017. [Online; accessed 12-October-2017].

[110] Pothier, Guillaume, Tanter, Éric, and Piquer, José M. Scalable omniscient debugging. In *Proceedings of the 22nd Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2007), pp. 535–552.

[111] Powers, Bobby, Vilk, John, and Berger, Emery D. Browsix: Bridging the Gap Between Unix and the Browser. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), pp. 253–266.

[112] Ratanaworabhan, Paruj, Livshits, Benjamin, and Zorn, Benjamin G. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In *USENIX Conference on Web Application Development* (2010).

[113] Raychev, Veselin, Vechev, Martin T., and Sridharan, Manu. Effective race detection for event-driven programs. In *Proceedings of the 2013 International Conference on Object Oriented Programming Systems Languages & Applications* (2013), pp. 151–166.

[114] Robinson, D., and Coar, K. The Common Gateway Interface (CGI) Version 1.1. RFC 3875, RFC Editor, October 2004. `http://www.rfc-editor.org/rfc/rfc3875.txt`.

[115] Rudafshani, Masoomeh, and Ward, Paul A. S. LeakSpot: Detection and diagnosis of memory leaks in JavaScript applications. *Software: Practice and Experience 47*, 1 (2017), 97–123.

[116] Sen, Koushik, Kalasapur, Swaroop, Brutch, Tasneem G., and Gibbs, Simon. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2013), pp. 488–498.

[117] Smedberg, Benjamin. NPAPI Plugins in Firefox. `https://blog.mozilla.org/futurereleases/2015/10/08/npapi-plugins-in-firefox/`, 2015. [Online; accessed 16-July-2018].

[118] Smith, Jerry. Moving to HTML5 Premium Media. `https://blogs.windows.com/msedgedev/2015/07/02/moving-to-html5-premium-media/`, 2015. [Online; accessed 16-July-2018].

[119] Stevenson, Michael. Perl and the birth of the dynamic web. `https://opensource.com/life/16/11/perl-and-birth-dynamic-web`, 2018. [Online; accessed 10-August-2018].

[120] Stroop, John Ridley. Studies of interference in serial verbal reactions. *Journal of Experimental Psychology 18*, 6 (1935).

[121] Tang, Shuo, Mai, Haohui, and King, Samuel T. Trust and Protection in the Illinois Browser Operating System. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), pp. 17–31.

[122] The Internet Archive. The Internet Archive Software Collection. `https://archive.org/details/software`, 2018. [Online; accessed 18-July-2018].

[123] The jQuery Foundation. jQuery. `https://jquery.com/`. [Online; accessed 30-April-2017].

[124] Topic, Dalibor. Moving to a Plugin-Free Web. `https://blogs.oracle.com/java-platform-group/moving-to-a-plugin-free-web`, 2016. [Online; accessed 16-July-2018].

[125] Undo. Reversible Debugging Tools for C/C++ on Linux and Android. `http://undo.io/`. [Online; accessed 30-April-2017].

[126] University of Illinois. Code Moo – A playful way to learn programming. http://www.codemoo.com/, 2015. [Online; accessed 18-July-2018].

[127] Vilk, John. [Browser Client] Minor memory leak: ".mail_source" event resubscriptions. https://github.com/mailpile/Mailpile/issues/1911, 2017. [Online; accessed 8-November-2017].

[128] Vilk, John. [Browser Client] Minor memory leak: Text nodes in notification area. https://github.com/mailpile/Mailpile/issues/1931, 2017. [Online; accessed 8-November-2017].

[129] Vilk, John. Fix memory leak in Element.removeData(). https://github.com/DmitryBaranovskiy/raphael/pull/1077, 2017. [Online; accessed 8-November-2017].

[130] Vilk, John. Fix memory leaks in data table / jqplot. https://github.com/piwik/piwik/pull/11354, 2017. [Online; accessed 8-November-2017].

[131] Vilk, John. Fix multiple memory leaks in UserCountryMap. https://github.com/piwik/piwik/pull/11350, 2017. [Online; accessed 8-November-2017].

[132] Vilk, John. Fix UIControl memory leak. https://github.com/piwik/piwik/pull/11362, 2017. [Online; accessed 8-November-2017].

[133] Vilk, John. JavaScript Memory Leak: #columnPreview click handlers. https://github.com/piwik/piwik/issues/12058, 2017. [Online; accessed 8-November-2017].

[134] Vilk, John. JavaScript Memory Leak: widgetContent $destroy handlers. https://github.com/piwik/piwik/issues/12059, 2017. [Online; accessed 8-November-2017].

[135] Vilk, John. Memory Leak: gtm.js repeatedly appends conversion_async.js to head when pushing to dataLayer. https://goo.gl/WFPt4M, 2017. [Online; accessed 6-November-2017].

[136] Vilk, John. Memory Leak in Preview Component. https://github.com/devtools-html/debugger.html/issues/3822, 2017. [Online; accessed 8-November-2017].

[137] Vilk, John. Memory Leak: material_select never removes global click handlers. https://github.com/Dogfalo/materialize/issues/4266, 2017. [Online; accessed 8-November-2017].

[138] Vilk, John. Minor frontend memory leaks due to unremoved LokiJS dynamic views. https://github.com/loomio/loomio/issues/4248, 2017. [Online; accessed 8-November-2017].

[139] Vilk, John. Minor JavaScript Memory Leak: piwikApiService allRequests array. https://github.com/piwik/piwik/issues/12105, 2017. [Online; accessed 8-November-2017].

[140] Vilk, John. Small Memory Leak and Correctness Bug in analytics.js. `https://issuetracker.google.com/issues/66525724`, 2017. [Online; accessed 6-November-2017].

[141] Vilk, John. Small memory leak: Callbacks added to window._xdc_ are never cleared. `https://issuetracker.google.com/issues/66529186`, 2017. [Online; accessed 6-November-2017].

[142] Vilk, John. Small Memory Leak in $rootScope.$on. `https://github.com/angular/angular.js/issues/16135`, 2017. [Online; accessed 8-November-2017].

[143] Vilk, John, and Berger, Emery D. Doppio: Breaking the Browser Language Barrier. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), pp. 508–518.

[144] Vilk, John, and Berger, Emery D. BLeak: Automatically Debugging Memory Leaks in Web Applications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2018), pp. 15–29.

[145] Vilk, John, and Berger, Emery D. BLeak repository. `https://github.com/plasma-umass/bleak`, 2018. [Online; accessed 20-March-2018].

[146] W3C. HTML5 Standard. `https://www.w3.org/TR/html50/`, 2014. [Online; accessed 27-August-2018].

[147] W3C. w3c/webidl2.js: WebIDL parser. `https://github.com/w3c/webidl2.js`, 2018. [Online; accessed 24-July-2018].

[148] Wheeldon, Brent. Unbind events to prevent memory leaks. `https://github.com/jeff-collins/ment.io/pull/138`, 2017. [Online; accessed 8-November-2017].

[149] Xia, Bingying. PacMan. `https://github.com/bxia/Javascript-Pacman`, 2013. [Online; accessed 30-April-2017].

[150] Xu, Guoqing (Harry), and Rountev, Atanas. Precise memory leak detection for Java software using container profiling. *ACM Transactions on Software Engineering and Methodology 22*, 3 (2013), 17:1–17:28.

[151] Xu, Min, Malyugin, Vyacheslav, Sheldon, Jeffrey, Venkitachalam, Ganesh, and Weissman, Boris. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Workshop on Modeling, Benchmarking and Simulation* (2007).

[152] Yang, Zhemin, Yang, Min, Xu, Lvcai, Chen, Haibo, and Zang, Binyu. ORDER: Object centRic DEterministic Replay for Java. In *Proceedings of the 2011 USENIX Annual Technical Conference* (2011).

[153] Zakai, Alon. Emscripten: an LLVM-to-JavaScript compiler. In *OOPSLA Companion* (2011), pp. 301–312.