Understanding Dynamic Behavior with Tools for Retroactive Investigation

Brian Burg

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2015

Supervisory Committee:

Michael D. Ernst, Chair

Andrew J. Ko, Chair

Steve Tanimoto

James Fogarty

Sean Munson

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Understanding Dynamic Behavior with
Tools for Retroactive Investigation

Brian Burg

Co-Chairs of the Supervisory Committee:
Associate Professor Michael D. Ernst
Computer Science and Engineering

Associate Professor Andrew J. Ko
The Information School

The web is a widely-available open application platform, where anyone can freely inspect a live program's client-side source code and runtime state. Despite these platform advantages, understanding and debugging dynamic behavior in web programs is still very challenging. Several barriers stand in the way of understanding dynamic behaviors: reproducing complex interactions is often impossible; finding and comparing a behavior's runtime states is time-consuming; and the code that implements a behavior is scattered across multiple DOM, CSS, and JavaScript files.

This dissertation demonstrates that these barriers can be addressed by new program understanding tools that rely on the ability to capture a program execution and revisit past program states within it. We show that when integrated as part of a browser engine, deterministic replay is fast, transparent, and pervasive; and these properties make it a suitable platform for such program understanding tools. This claim is substantiated by several novel interfaces for understanding dynamic behaviors. These prototypes exemplify three strategies for navigating through captured program executions: (1) by visualizing and seeking to input events—such as user interactions, network callbacks, and asynchronous tasks; (2) by retroactively logging program states and reverting execution back

to log-producing statements; and (3) by working backwards from differences in visual output to the source code responsible for inducing output-affecting state changes. Some of these capabilities have been incorporated into the WebKit browser engine, demonstrating their practicality.

ACKNOWLEDGMENTS

The author thanks everyone for being awesome. TODO rewrite.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

The world wide web's rise as *the* universal runtime environment has democratized the development of documents, applications, and user interfaces. Unlike on most platforms, web programs are transmitted in source form as HTML, CSS, and JavaScript code. Using web developer tools included with most web browsers, a user can inspect and modify a client-side web program's source code and runtime states. This capability dramatically lowers technical barriers to finding interesting behaviors within third-party web content and inspecting the corresponding source code for a behavior.

Code that implements designs and behaviors is readily available on the web, but is just as difficult to understand and debug as code written for any other platform. Modern web programs are complex, interactive applications built using a combination of several tools, frameworks, languages, and other technologies. While originally intended for static documents, web technologies such as HTML, DOM, JavaScript and CSS are now the building blocks for large cross-platform application suites developed by teams of software engineers. Beyond the challenges inherent to building user interfaces, the interactions between web technologies introduce significant incidental complexity. The combination of declarative CSS styles, imperative JavaScript code, and a retained document rendering model can obfuscate a program's dependencies and causal relationships.

Existing developer tools are inadequate for debugging a web program's interactive behaviors. Many tools distributed with web browsers simply reimplement "baseline" tools from other development environments, such as logging, profiling, and breakpoints. These tools are ill-suited for understanding nondeterministic, interactive, and time-sensitive behaviors that are pervasive in client-side web programs. The complexity of web pro-

grams is often limited by developers' ability to debug and maintain these systems with poorly-suited tools. Using application frameworks and code conventions can mitigate the incidental complexity of writing these programs; however, well-disciplined code can sometimes be even more difficult to understand at first.

1.1 The Problem

Several barriers stand in the way of understanding dynamic behaviors: reproducing complex interactions is often impossible; obtaining runtime states relevant to a behavior requires preëmptive tool configuration; and the code that implements a behavior is scattered across multiple DOM, CSS, and JavaScript files and difficult to locate statically. Underlying these issues is a common problem: *existing tools allow a user to inspect current or future program states in multiple executions, but not past program states from a single execution*. A user must configure debugging tools—such as breakpoints or logging—before the behaviors of interest actually occur. This places a considerable burden on the user: in order for the tools to have any benefit, she must find relevant code fragments without feedback and correctly predict what runtime states might be useful (and how to gather them). In the face of nondeterministic behavior, repeatedly reproducing the same behavior after changing tool configurations is error-prone or impossible. Even when behavior is deterministic, logging desired runtime states or pausing the debugger at an important statement requires careful planning and time-consuming experimentation. Debuggers, profilers, and other tools that support inspection of live runtime information during execution are often inappropriate for investigating interactive behaviors, because these tools may slow or suspend execution and thus spoil the behavior or the runtime data. Lastly, existing tools do not allow navigating backwards from causes to effects, i.e., from current program states to previously-executed code responsible for the state. Instead, a user must a user must predict cause-effect relationships responsible for program states, configure their tools to gather information at an earlier instant, and confirm their guesses by observing actual runtime behaviors.

1.2 Addressing the Problem

Existing tools are hamstrung: without access to past program states, they are necessarily limited to inspecting future states. What if a developer could gather, visualize and revisit *past* program states after they already occurred, rather than configuring tools for current and future program states? What if it were possible to capture a single execution and retrieve runtime states from it as needed, rather than guessing ahead of time what program states might be useful to log?

Consider the task of debugging an interaction in a video game: instead of manually reproducing game behavior whenever different runtime information is desired, a developer could play the game once and later go "back in time" in the captured execution to examine past runtime states. As the developer's understanding of the program grows, they can jump directly between relevant program states. This workflow avoids repetitious, error-prone gameplay and decouples playing the game from interruptions such as setting up logging, turning breakpoints on and off, or performance slowdowns from heavyweight instrumentation.

In this dissertation, I investigate how this *retroactive* approach to program understanding can be realized through novel runtime techniques, user interfaces, and integrations with new and existing developer tools. In particular, I claim the following thesis statement:

> *The ability to revisit past program states enables new tools for understanding dynamic behaviors in web programs, and browser engines can provide this capability through fast, transparent, and pervasive deterministic replay.*

I substantiate this claim by investigating two related lines of research: how browser engines can capture and replay web program executions using deterministic replay techniques; and what tools, strategies and interfaces are necessary to find relevant program states and facts within a captured execution. In order to revisit arbitrary past program

states during program understanding tasks, it must be possible to exactly reproduce an execution on demand. The first part of this dissertation investigates how deterministic replay techniques can provide this capability for high-level managed runtimes such as browser engines. In order to work with executions as first-class objects, it must be possible for a replay-enabled developer tool to reference specific execution instants and efficiently collect historical runtime data. The middle part of this dissertation develops several high-level programmatic interfaces that serve as a crucial linkage between the capabilities of a deterministic replay infrastructure and the needs of a replay-enabled tool. In order to act upon past program states, it must be possible to quickly find relevant facts among the vast amount of runtime information produced during an execution. The last part of this dissertation develops several task-oriented navigation strategies and supporting replay-enabled tools for finding relevant program states in a captured execution.

1.3 Definitions

This dissertation builds upon work from various disciplines and fields such as Human-Computer Interaction, Program Analysis, Compilers, and Software Engineering. Thus, it is useful to define and consistently use key terms that are otherwise prone to misinterpretation.

Many terms exist to categorize people who perform *programming*: instructing a computer (via code or other directives) to perform actions at a later time. This dissertation refers to any such person with the generic term *developer*. A *novice developer* has a little practice; a *skilled developer* has more than a little practice[1]; a *professional developer* is paid for his or her programming activities. An *end-user programmer* writes code only with the purpose of supporting a larger task or goal. A *tool developer* creates tools for use by *tool users*, who create programs for *end-users*.

This dissertation is primarily concerned with the family of tasks referred to as *program*

[1]Where possible, scenario-relevant modifiers such as *successful* and *unsuccessful* are preferred in place of subjective or demographic-based terms such as *senior*, *novice*, and *skilled*.

understanding: any process undertaken (typically by a developer) to develop an explanation of how a program did execute, will execute, or will not execute. Specific program understanding tasks include *feature location*: developing an explanation of what code implements a specific behavior; and *debugging*: developing an explanation of undesirable or unexpected behaviors. Various debugging and bug-related terms also deserve definition. A *failure* is an undesirable program output that does not conform to the program's expected behavior. A *fault* is a program state that may lead to a failure. This dissertation uses the terms *error* and *defect* interchangeably to refer to parts of program code that cause a *fault* and/or *failure*. A *bug* loosely refers to single or a combination of failure(s), fault(s), and/or defect(s).

The long history and evolution of the world wide web has led to many confusing, similar terms. *A web developer* is a developer who produces web content. This dissertation uses the term *web program*[2]to refer to any document consisting of HTML, CSS, DOM, JavaScript, and related technologies that are viewable by a *web browser*. A *web browser*—sometimes referred to as a *user agent* in web standards—is an end-user application for viewing web content. A *browser engine* is a managed language runtime capable of downloading, parsing, interpreting, and rendering untrusted web content, and is separate from other web browser functionality such as bookmarks, tabs, and other user interface elements. Finally, *web developer tools* are program understanding tools used by web developers. This dissertation mainly discusses web developer tools that are distributed as part of a web browser.

TODO define replay-related terms TODO define execution event similar to Andy's document

[2]In other contexts, web programs are also referred to as web pages, web content, web applications, and other terms to emphasize program characteristics such as complexity and interactivity. This document uses the single term *web program* and modifies it as necessary to convey the intended population of programs.

1.4 Contributions

This dissertation introduces several major contributions that extend the state of the art in runtime techniques and program understanding tools:

- Techniques for fast, pervasive and transparent deterministic replay of web content.
- Algorithms for revisiting any executed statement within a captured execution.
- Two visualizations of a captured execution that support navigating via top-level actions.
- An interface for retroactively logging runtime states and revisiting their execution context.
- The first user study examining the benefits, drawbacks, and design concerns for interactive record/replay user interfaces.
- Algorithms for efficiently detecting, serializing, and comparing visual states over time.
- Algorithms for establishing causality between visual changes, state changes, and code.
- An interface for feature location based on comparing output and state changes.

1.5 Outline

The remainder of this dissertation is organized as shown in Figure 1.1. Following this section is Chapter 2, which surveys prior work with a focus on techniques, tools, interfaces, and information needs that inform the above contributions. The second part describes the Dolos replay infrastructure (Chapter 3) and extensions to Dolos that support common

replay use cases (Chapter 4). The third section describes a series of program understanding tools that support different navigation strategies: navigation via inputs (Chapter 5), navigation via logged outputs (Chapter 5), and navigation via visual state changes (Chapter 6). Chapter 7 presents the first user study exploring how replay interfaces are used during debugging tasks. The final part of this document sketches several directions for future research (Chapter 8) and presents the conclusions of this research (Chapter 9).

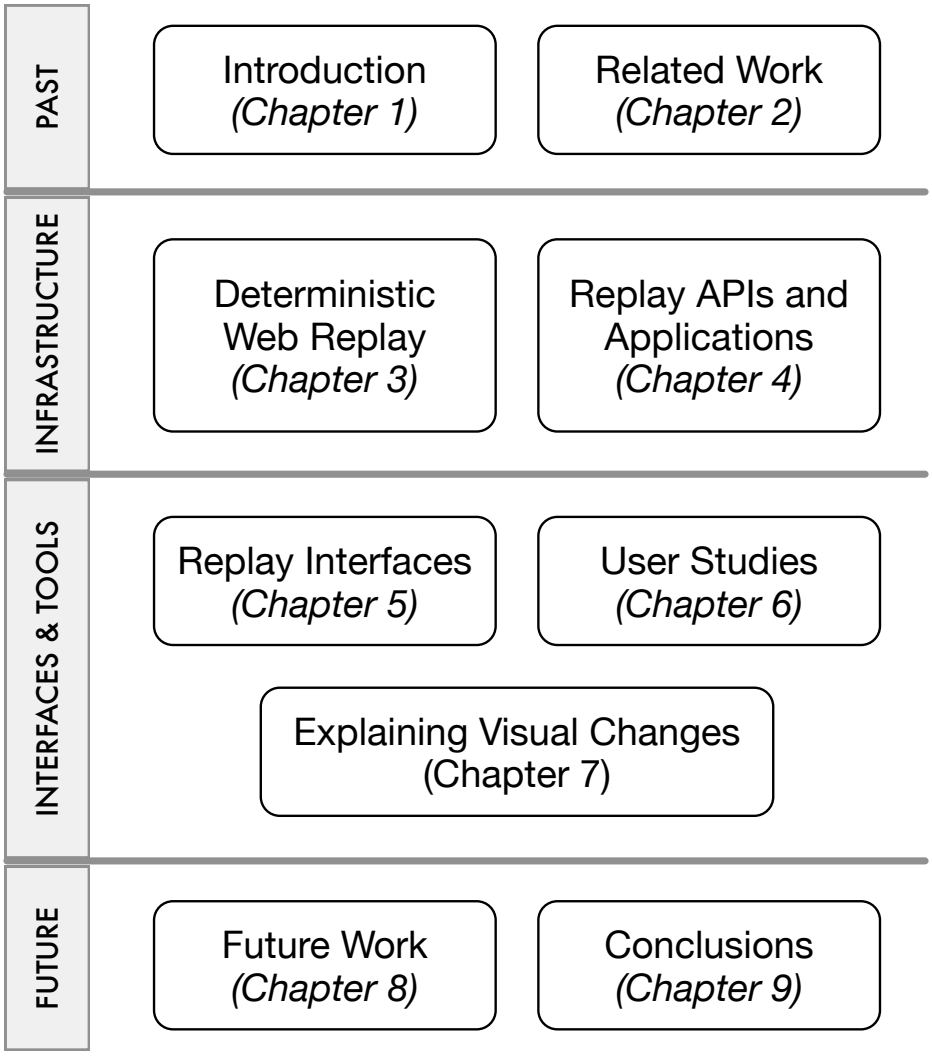

Figure 1.1: Major parts and chapters of this dissertation. TODO Move studies to last.

Chapter 2
RELATED WORK

2.1 Reproducing and Revisiting Dynamic Behavior

A key enabling ability for understanding dynamic behavior is the ability to review past program states. Existing approaches to recreating program states can be divided into two main strategies. Deterministic replay techniques re-execute a captured program execution to create program states. Post-mortem techniques log a detailed trace of program operations and reconstruct state by simulating relevant operations.

2.1.1 Deterministic Replay

Deterministic replay is a well-studied technique [31, 39] for recreating or replicating a specific execution. Contemporary research focuses on techniques for low-overhead replay of multithreaded and multicore programs; virtual machine [40] and hardware support for capture and replay; adding replay support to new languages and runtimes; and novel applications of replay techniques to other domains [26, 94]. Few of these systems are production quality, and even the most robust and widely-deployed replay tools [91, 138] are designed for expert use via debugger commands (discussed in Section 2.1.3).

Prior research into deterministic replay for web applications has mostly focused on standalone tools for cross-browser replay. Mugshot [88] uses a reverse network proxy to instrument incoming web applications at the source level. It injects native function wrappers and DOM event listeners into the web application to transparently capture and replay JavaScript from "user-space". JSBench [112] uses the same instrumentation approach to synthesize JavaScript benchmarks from a captured execution. FireCrystal [98] is an extension for Firefox [90] that captures DOM events and attempts to replay them

on an isolated copy of the web application. The WaRR [5] infrastructure captures DOM events from within WebKit and replays them through a Selenium [132] plugin.

Dolos differs from these tools by choosing integration with standard developer tools [97, 142] and high-fidelity replay as top-level design goals. Native tool integration is directly at odds with cross-browser approaches because of the sandboxed nature of web applications. A sandboxed JavaScript program (using GDB terminology [44], an *inferior process*) cannot interact with privileged JavaScript debuggers (a *superior process*), so a replay system implemented in the inferior process is unusable while the program is paused at a breakpoint. High-fidelity replay of JavaScript is not possible with "user-space" libraries and source instrumentation because not all APIs can be mediated, and many sources of nondeterminism—such as timers, animations, and resource loading—cannot be controlled from outside of the rendering engine.

but only recently have researchers investigated replaying interactive, user-driven programs in ways that produce full visual output.

While Dolos goes to great lengths to make JavaScript execution completely deterministic, this may limit a user's wish to reuse nondeterministic inputs on a different program version [26, 51, 61, 94] or intentionally diverge a replayed execution to explore alternate histories. Probe points (Chapter 5) address logging—the most common reason to alter the program—but are not designed for side-effecting operations. Scribe [70] is a multicore deterministic replay system designed to "go live" from the end of a captured execution; Dora [136] extends Scribe to meaningfully exclude portions of a recording that are affected by divergence. Another line of work attempts to deterministically reproduce outputs from anonymized logs [30], partial runtime state [59], or minimal replay logs [2].

2.1.2 Tracing and Post-mortem Techniques

A well-studied alternative to deterministic replay systems are post-mortem trace-based tools. These tools gather exhaustive traces of execution at runtime and provide affor-

dances for querying, analyzing, or visualizing the traced behavior after the program has finished executing. We discuss several trace-based systems that are designed for program understanding tasks.

Trace-based techniques are generally avoided for JavaScript programs because the output of web applications is highly visual in nature and capturing a trace in memory can quickly make an application become unusably slow. Two exceptions are JSMeter [108] and DynJS [111], which both instrument the browser itself to collect a detailed trace of JavaScript execution for offline simulation.

Amber [95] and Nirvana [12] are two tools for efficiently collecting, storing, and indexing traces from executions of x86 binaries. Both tools use dynamic binary rewriting frameworks like Valgrind [93] to instrument and capture a detailed log of register operations, and memory operations, and control flow. Nirvana employs sophisticated compression mechanisms and partial coalescing to achieve low overhead while the program executes, but must re-simulate execution to produce accurate results. Amber incurs high capturing overhead but precomputes indexes of memory effects and requires no re-execution. Tralfamadore [79] captures a low-level unindexed trace at runtime; offline, it runs a dynamic analysis over the trace in a streaming pipeline that successively transforms the trace into higher-level events that are meaningful to a user.

ODB [81] was the first trace-based omniscient debugger for Java programs. It heavily instruments JVM bytecode to record a trace of all memory activity and control flow. It uses information in the trace to populate IDE views containing value histories, rematerialized local variables, and a call stack. The omniscient debugger TOD [106] improves on ODB by incorporating modern database indexing and query optimization techniques and partial deterministic replay. STIQ [105] further improves performance with optimizations for random memory inspection, causality links between reads and writes, and bidirectional debugger commands (Section 2.1.3). In order to recreate visual output for post-mortem debugging, Whyline [65] uses domain-specific instrumentation of GUI toolkit APIs in order to save a trace of relevant toolkit invocations.

2.1.3 *Navigating via Debugger Commands*

Text-based commands are often the *only* interface for controlling back-in-time debuggers or deterministic replay infrastructures [91, 137, 138], and often supplement visual interfaces [81]. In 1990, Tolmach and Appel [134] first described the `reverse-step` and `reverse-continue` commands in a debugger for the ML language. The seminal work for imperative debuggers is Boothe's description of efficient counter-based bi-directional debugging algorithms [17], which includes commands for `reverse-step-into`, `reverse-step-out`, `reverse-continue`, and `reverse-watchpoint`. Arya et al. have recently proposed [7] algorithmic improvements to `reverse-watchpoint` based on decomposing large debugger commands [137] like `continue` into a sequence of `step-over` and `step-into` commands. Dolos does not implement any of these commands, but there are no known complications to doing so.

TODO What does rr do?

2.1.4 *Navigating via Program Outputs*

Timelapse's use of timelines and seekable outputs is specifically designed for casual use during program understanding tasks. It draws on a long tradition [53] of graphical history visualizations such as those used extensively in the Chronicle [47] tool. Few deterministic replay tools can seek execution directly to specific logged outputs without auxilliary use of breakpoints. The YingYang [86] live programming system is one exception; however, it depends on a restricted programming model where all operations are undoable and commutative, and does support interactive programs. Web browsers and other high-level application platforms are able to relate rendered UI elements to their corresponding source implementation or runtime objects, but this is typically limited to the currently visible output of the program.

DejaVu [61] combines affordances for replaying, inspecting, and visualizing a computer vision kernel program over time. It decouples video inputs from outputs so new

versions of the kernel can be tested against the same inputs. DejaVu assumes a deterministic, functional kernel so that "checkpointing" and replaying the program is a matter of re-executing the kernel from a specific frame of the input stream.

Post-mortem, trace-based investigation tools such as Whyline [65], TOD [106], and ODB [81] support navigating through traces by selecting simulated visual output, console output, or historical values of objects. Rather than using selections as a target for re-execution, these tools search for the selected instant over a large trace, and display it in the context of related information such as a matching call stack or local variables.

2.2 Analyzing and Understanding Dynamic Behavior

Scry and Overscan draw on a vast literature of techniques for instrumenting, analyzing, and visualizing runtime data for program comprehension [33]. We summarize the most relevant work below to clarify our chosen points in the design space, and describe alternatives and their tradeoffs.

2.2.1 Implementing and Specifying Dynamic Analyses

The ways in which instrumentations and dynamic analyses code are specified and implemented varies widely between languages, toolchains, and application domains. We are particularly concerned with two aspects: the *instrumentation mechanism*—how a base program is modified to gather data—for its effect on performance and compatibility, and the *specification mechanism*—how an analysis defines its instrumentation needs—which greatly affects the complexity of a particular dynamic analyses.

In the domain of JavaScript and web applications, the vast majority of research tools [1, 87, 88, 96, 112, 122, 133] are implemented using source-to-source transformations (also referred to as "source instrumentation") because there is no standardized, cross-platform bytecode for JavaScript. Instrumentation is added by intercepting, parsing, and modifying JavaScript source before it reaches the browser using reverse proxies and JavaScript

AST libraries . Source instrumentation is used to install API shims over native methods, add instrumentation hooks around statements or expressions, and interpose on DOM events.

Source instrumentation works for small examples and research prototypes, but its drawbacks are too severe for the approach to be used as a basis for interactive, integrated tools that support program comprehension. Transformed source code is effectively obfuscated, rendering the code unusable by source code editors and debuggers that cannot distinguish application code from instrumentation code. Control flow, allocations, and other dynamic behaviors are also perturbed because instrumentation and application code execute at the same level. Source instrumentation incurs high overheads at instrumentation time and runtime, and is ill-equipped to handle the dynamic features of JavaScript [111] such as `eval` [113] and aliasing of native methods.

In other runtime environments, bytecode instrumentation [9, 65] and dynamic binary translation [12, 93] are the standard mechanisms for modifying programs for analysis purposes. Bytecode instrumentation of JavaScript programs is still relatively uncommon. Though JavaScript bytecode instrumentation is inherently browser-specific and more difficult to implement and maintain, it has significantly better performance and can gather data from the virtual machine that is unavailable to JavaScript code [13, 111, 113]. Bytecode instrumentation and dynamic binary rewriting can be made compatible with debuggers, profilers, and other tools. For example, Valgrind implements an in-process remote debugging server [107] for GDB which translates debugger commands to work on instrumented code and additionally exposes values of shadow memory and shadow registers.

Dynamic analysis and instrumentation frameworks [9, 12, 84, 93, 122, 133] provide rich APIs which abstract away the significant implementation complexities of instrumentation mechanisms. Generally speaking, frameworks provide a discrete set of instrumentation callbacks (memory read/writes, function call/return, system calls, allocations, etc) or they provide a declarative API for mutating specific AST locations or bytecode sequences. Overscan does not emit events that can be expressed as instrumentation hook

pointcuts; other instrumentation events (DOM events, timer registrations, etc.) can be obtained by analyses through built-in instrumentation of the browser platform.

Aspects have been used to declaratively specify instrumentation, especially for coarse-grained analyses in high-level languages such as Java [104, 106, 117] and JavaScript [80, 133]. However, aspects were not designed for instrumentation, so aspect languages have been extended to simplify common tasks like instrumenting basic blocks [37], sharing state across join points [84], and cheaply accessing static and dynamic context at runtime. For greater power, analyses must resort low-level bytecode manipulation libraries [9] to suit their needs. Overscan uses a declarative API for specifying instrumentation hook positions in an AST; the actual data collection and analysis code is implemented by callbacks written in C++. Scry's analyses cannot modify the JavaScript source, and should not perform side-effecting operations such as mutating the JavaScript heap.

Shadow values—duplicated program values that exist in a parallel address space for instrumentation purposes—are a powerful mechanism for implementing complex online dynamic analyses. A dynamic analysis (only one) can add and modify custom annotations in the shadow values as the program accesses the corresponding program values. Valgrind [93] implements shadow registers and shadow memory for x86 binaries. Jalangi [122] implements shadow values for JavaScript by wrapping objects within user-specified segments of JavaScript into a tuple of the application value and the shadow value. Uninstrumented code uses normal application values. ShadowVM [85] is an asynchronous, isolated VM that runs analysis code in a separate thread or process. In addition to providing shadow values, it provides strong guarantees of isolation and allows an analysis to be profiled and optimized independently of the target program.

2.2.2 Composable and Scoped Dynamic Analyses

Most instrumentation frameworks (with the notable exception of DTrace [25]) are not dynamic: they assume that only one analysis is active at any given time, that the set of active

analyses is constant over the program's execution, and that instrumentation is applied equally to all code. Some researchers have investigated ways of making instrumentation more composable, dynamic, and scoped, mainly in the context of Aspects. Ansaloni et al. [6] discuss recent work and open problems in this space, using a running example of three composed dynamic analysis: a calling context profiler, a basic block profiler, and an allocation profiler. We plan to use a similar case study to informally validate the design of Overscan's instrumentation affordances and support for dynamic, scoped instrumentation.

Composable instrumentation requires a framework to mediate the interactions between several competing instrumentations. Clearly, naively instrumenting low-level bytecode or source code is not composable because there's no way to distinguish instrumentation from client code. Researchers of aspect languages have long been concerned with unexpected interplay between aspects that use the same join points, and have developed mechanisms to stratify execution [130, 131] so that aspects cannot advise other aspects. Overscan provides a declarative instrumentation API that multiplexes instrumentation hooks for several active analyses [89], but executes instrumentation code in C++ partially to avoid perturbing execution and introducing meta-levels.

To limit the extent of instrumentation and data collection, prior work relies primarily on static and dynamic context to selectively instrument code or collect data. Overscan's mechanisms for scoping based on time and static scope are inspired by Reflex's support for spatial and temporal filters of behavioral reflection [129], as well as tools that add and remove dynamic instrumentation to running programs [25, 101, 110].

2.2.3 Retroactive and Post-mortem Analysis

Several lines of work have investigated techniques for running a dynamic analysis "offline" by decoupling the analysis from a live execution. Overscan uses the strategy of running dynamic analyses on a replayed execution [28, 120], which we refer to as retroac-

tive analysis. More common in the literature is the strategy of post-hoc trace querying [45, 95, 143] and analysis [78, 104, 105, 106, 108, 111], which we collectively refer to as post-mortem analysis because a live execution is not necessary to perform the analysis. Several projects [12, 29, 105, 122, 143] combine trace-based, query-based and replay-based approaches to achieve interactive response times for common back-in-time queries. The common idea is to save only an index of a program trace's activity, and perform partial replay from a checkpoint to re-materialize a full-fidelity execution trace when necessary.

2.2.4 Visualizing Dynamic Behaviors

Visualization is a large field with many applications to program comprehension. We focus our attention on visualizations of dynamic behaviors, static and dynamic control flow, live execution, and ways in which visualizations are incorporated in development environments and developer workflows.

TODO cite OptimizationCoaching (Racket and JS), LhotakL2004, KoCrystal

TODO cite Orso/Harrold papers that describe some ugly encodings

At the lowest level, many tools visualize dynamic behavior of specific expressions and statements by augmenting source text with overlays, background and foreground color shading [83], context menus [65, 118], inline gutter/scrollbar widgets [83, 118], runtime values [65, 83], or links to other views with information about specific instances [65, 81]. These lightweight visualizations can be used to highlight multi-line units of code, but this can quickly become unmanageable in the presence of namespaces, anonymous event handlers, and other language features that cause definitions and side-effecting statements to be frequently juxtaposed. For example, if two functions are nested in JavaScript, it is unclear whether a statement highlighted in the inner function represents execution of the inner function or instantiation of the inner function as the outer function execution. Scry handles this situation by using visualizing both notions (**??**) and using hints about the static source structure to explain scoping relationships.

Visualizations of control flow or causality must relate many source elements scattered throughout code that cannot fit into a single source editor view. Graph-oriented visualizations [73] and sequence diagrams [65] are common ways of showing these relationships, but must be used carefully to avoid overwhelming the user with irrelevant information. Like Reacher [73] and Whyline [65], Scry addresses this challenge by restricting visualization to only those relationships that the user manually expands. Graphs and sequence diagrams are inherently distinct in form from source code, so visualizations must include contextual hints (such as hyperlinks or a call stack) to remind the user of the context of each source element. Scry takes this idea one step further by embedding a source editor into sequence diagrams (**??**) in order to show additional dynamic context around each source element.

An important dimension in visualizing dependencies and relationships is spatial organization: how elements are arranged in space, and how this arrangement implicitly conveys relationships. For example, a timeline of multithreaded execution [135] instantly conveys temporal dependencies among events generated by different threads (even if these are not necessarily accuarate). A call stack visualization [48] can exploit the convention of a stack growing downward to implicitly connote the relationship between callers and callees. Scry uses both of these conventions (along with explicit arrows linking related source elements [65]) to distinguish synchronous caller–callee relationships and asynchronous registration–invocation relationships (**??**). This design is influenced by the VIVIDE programming environment [128], which integrates static and dynamic views of a program on an infinitely horizontally scrolling tape of connected editor windows. Lastly, the Code Canvas line of work [18, 35, 36] explores an infinite two-dimensional pan-and-zoom canvas for arbitrary spatial organization of editors and runtime state. This approach is promising for sharing code investigations with others, but seems difficult to integrate with standard IDE conventions and can become cluttered. The Light Table IDE[1] originally supported arbitrary positioning of editors on a single canvas, but has since reverted back

to the dominant tabbed editor interface.

Developers frequently switch among multiple levels of detail and abstraction to better suit their information needs. SHriMP Views [126] were an early exploration of providing multiple levels of detail within the same editor. Other research has used multiple levels of detail to explain cauasality relationships for JavaScript events [1]. Scry primarily exposes multiple levels of detail through progressive data collection. For example, **??** shows several levels of execution profiling. Some visualization tools specifically target discovery of high-level trends over the entire execution [34, 42, 58, 106, 116, 135]. Scry uses low fidelity runtime data from the entire execution to fade out unexecuted code, but relies on the Web Inspector's built-in timeline for discovery of higher-level trends.

2.3 Designing Tools For Program Comprehension

While the technical aspects of implementing low-overhead replay, instrumentation and analyses are challenging and well-studied, their value to a developer ultimately hinges on the effectiveness of the tool with which they interact. Every tool developer hopes that their tool is effective, so why do some tools have a large impact, while others are never used? Researchers in fields such as psychology, sociology, HCI, ergonomics and computer science have developed several theories and models to account for program comprehension and tool use from a cognitive perspective. In this section, we connect these developments to more recent research that focuses on characterizing developers' information needs, and how these questions motivate comprehension tool research.

2.3.1 Cognitive Models of Comprehension and Tool Use

Researchers have long sought to understand the cognitive mechanisms that underly program comprehension and related activities such as debugging. Détienne [38] provides a comprehensive history of cognitive models of program comprehension. Researchers

[1] **https://lighttable.com**

originally modeled program comprehension as a monolithic activity patterned after text comprehension, using concepts such as chunking, top-down and bottom-up comprehension to account for developer's various strategies for reading code. von Mayrhauser and Vans's integrated meta-model [139] is representative of influential cognitive models from the 1980's and early 1990's. Storey, Storey et al. provides an insightful catalog of cognitive design elements and design implications for visualization and tool design that arise from these major theories.

In his dissertation, Walenstein [140] adapts the theory of distributed cognition [56] to the domain of software engineering to model exactly how comprehension tools become *useful*. He focuses specifically on the ways in which the cognitive tasks of software development are reconfigured and redistributed by the introduction of developer tools. Using this framing, he proposes to judge usefulness of a developer tool on the basis of how it is able to redistribute cognition between the tool and the developer. For example, by keeping a navigable history of search results, an IDE can offload the significant cognitive effort that would be required for the developer to maintain the same history.

2.3.2 Information Needs and Developers' Questions

In writing about cognitive questions and design elements for software visualizations, Petre et al. [102] raise critical questions about the purpose, design, and interpretation of visualizations. They argue that tool designers must know what programmers actually *do* and ask in practice, so that visualizations and other tools support rather than conflict with these natural representations. Hence, researchers have focused on understanding common modes of developing software [67, 75], collaborating, seeking information [19, 54], describing implicit knowledge [27], and questions during software maintenance [123]. While cognitive models tend to abstract away from detailed examples of information, tool builders necessarily must design for specific use cases and capabilities. Programmers' questions are the crucial link between cognitive models of comprehension and tools that

can enhance a programmer's capabilities. Regardless of the specific theory of model of program understanding, all models require information—whether as evidence that tests a hypotheses, as data that solidifies mental models, or as a way to reflect and make explicit the implicit boundary of what the programmer does and does not know.

In the past 20 years, researchers have shifted from developing large-scale cognitive models and theories to investigating specific aspects of development. While developing the Integrated Meta-Model, von Mayrhauser and Vans [139] began making connections between cognitive models, program understanding tasks and subtasks, and specific information needs formulated as questions. These information needs were gathered from a talk-aloud protocol as part of a study wherein professional developers fixed a bug.

Since von Mayrhauser and Vans's original study, other researchers have used similar study designs to understand developers' practices during code navigation [76, 77] and information-seeking [19, 66, 103, 123], and problem-solving strategies. Most relevant to this dissertation, researchers have catalogued common types of questions, including reachability questions [71, 74], hard-to-answer questions [72] and questions about output and causality [65]. These questions form a comprehensive account of a tool's capabilities from the perspective of its users; many tool papers (including this thesis proposal) begin their motivation by considering how these questions could be answered.

2.4 Working from Visual Output to Runtime States and Code

2.4.1 Linking Visual Output to Program States

Crystal, Whyline

2.4.2 Tracing Program States to Responsible Code

2.4.3 Minimizing and Slicing Dependencies

2.4.4 Visualizing State Differences

Chapter 3

DETERMINISTIC REPLAY FOR WEB PROGRAMS[1]

Many program understanding questions can be answered with runtime states [63, 72, 123], but reproducing and gathering runtime states is difficult with existing tools. A developer must employ an iterative process of configuring developer tools, reproducing behavior with tools enabled, and then using the tool-collected runtime states to inform further tool use. This iteration requires the developer to repeatedly reproduce the behavior they are inspecting, which can be time-consuming and error-prone [149]. In the case of interactive programs, even reproducing a failure can be difficult or impossible: failures can occur on mouse drag events, be time-dependent, or simply occur too infrequently for a developer to easily reach a program state suitable for debugging.

What if it were possible to deeply examine program states from a single execution, rather than states from a family of similar executions? A developer would need to manually reproduce behavior just once. Runtime states which were previously hard to isolate would be just a breakpoint or `print` statement away. A developer could work backwards from effects back to causes [63] without intermediate manual reproduction steps.

Two families of techniques allow for a developer to inspect past program states at a later time: tracing and deterministic replay. Tracing (Section 2.1.2) is a superset of adding ad-hoc `print` statements to a program. Essentially, if enough `print` statements were added to a program, then any program state could be reconstructed from logged output. Deterministic replay (Section 2.1.1) is a technique that can capture a single program execution as it executes, and then automatically re-execute it repeatedly and automatically, without requiring manual user interaction.

---

[1] Contributions in this chapter are described in part in Burg et al. [22].

Deterministic replay techinques have great potential, but have not been widely adopted yet. The choice of deterministic replay over tracing is critical to the goal of improving existing developer tools, which generally operate on live executions rather that serialized program states. However, prior deterministic replay approaches have major shortcomings: they not integrate well with breakpoints, logging, and other developer tools; they often have poor performance; and they do not have sufficient fidelity to replay complex web programs.

This chapter describes Dolos, a novel deterministic replay infrastructure for web programs that addresses shortcomings of prior work. To ensure deterministic execution, Dolos captures and reuses user input, network responses, and other nondeterministic inputs as the program executes. It does this in a purely additive way—without impeding the use of other tools such as debuggers—via a novel adaptation of virtual machine deterministic replay techniques to the execution environment of web programs.

3.1 Background

This section introduces important background concepts: what web programs are, how rendering engines execute web programs, how rendering engines and browsers are architected, and how this impacts deterministic replay.

3.1.1 Web Programs

Web programs are event-driven, interactive, and highly visual programs typically downloaded in source form over a network connection. Once its resources are parsed and evaluated, a web program's execution is driven by user input, asynchronous tasks, network traffic, and other events. Interactions are programmed using JavaScript, an imperative, memory-safe, dynamically-typed scripting language. Web programs make extensive use of platform APIs to access persistent state, make network requests, programmatically render visual output.

The execution model of web programs mirrors that of typical GUI frameworks. Work performed by web programs is scheduled cooperatively in a single-threaded event loop, and execution is naturally divided into a series of *event loop turns*. Worker threads can perform parallel computation and communicate with the main program via message passing. A web program can communicate with other web program instances via message passing, but are otherwise isolated from other contexts.

As an evolution of a static document format, web programs do not have explicit boundaries of scope and extent like those associated with processes in modern operating systems. Web programs can embed sub-programs inside `<iframe>`, `<frame>`, and `<svg>` elements; the transitive tree of web programs is referred to as a program's *frame tree* (collectively, a *page*). The means of communication between parent and child programs in the frame tree depends on their origins. In some cases they can directly access each other's heap data, and in other cases they may only communicate via message passing. To simplify the situation for the purposes of deterministic replay, we consider the scope of a single execution to encompass an entire frame tree. An execution begins when the root node of the frame tree (hereafter, the *main frame*) initiates a navigation to a new document. An execution ends when the main frame initiates a navigation to a different document. Thus, the extent of a single execution is between two navigations of the main frame.

3.1.2 Rendering Engines

The core functionality of a web browser is referred to as a *rendering engine*. Rendering engines are complicated execution environments that produce a web program's visual output. A rendering engine processes inputs from the network, user, and timers; executes JavaScript; computes page layout; and renders output to the screen. The rendering engine schedules asynchronous computation using a cooperative, single-threaded event loop. Features that are ancilliary to a web program's execution, such as bookmarks and a browser's address bar, are provided by browser applications instead of the rendering

engine.

3.1.3 Features, Ports, and Platforms

Rendering engines are often shared among multiple operating systems (hereafter, *platforms*), browsers and developer SDKs (hereafter, *ports*). For example, the WebKit rendering engine is used by the Cocoa toolkit (Mac and iOS), GTK toolkit (Mac OS X, Windows, Linux), and EFL toolkit (Enlightenment); the Blink rendering engine is used by the Chrome browser (Mac OS X, Windows, Linux), Opera browser, and many other applications. Some rendering engine ports expose separate public APIs, which allow external programs (hereafter, *embedders*) to embed the rendering engine and customize its settings and behavior in predefined ways.

How a web program executes is highly dependent on the specific rendering engine used to execute it. To support a variety of uses, rendering engines aggressively modularize capabilities and features, allowing some features to be enabled, disabled, or customized at compile-time or runtime. Most JavaScript-accessible APIs are standardized, but many lack common test suites, leading to subtle interoperability issues. Many rendering engines expose nonstandard APIs, new input modalities, and embedder-specific functionality. Web programs often perform *feature detection*—programmatically testing for existence of specific features—and alter their execution based on their execution environment.

3.1.4 Browser Architecture

Modern browser architectures [109] are designed primarily with performance and security concerns in mind. Servo [4], WebKit, and Blink/Chromium use a multi-process model to enforce least privilege, isolate different web programs, and provide coarse-grained parallelism. Low-privilege tasks such as compositing, networking, rendering web content, tool interfaces, and persistent state storage run in their own child processes and commu-

nicate with a parent process via message-passing. Each child process is isolated using operating system sandboxing; if one child process crashes due to a bug or vulnerability, other processes are unaffected.

Multi-process architecture has several implications that make deterministic replay easier to achieve. Strict interfaces at process boundaries help to reveal potentially nondeterminisic data flows between major browser and engine components. Messages between processes cannot reference shared mutable state, so they must fully and exactly characterize inputs which are often nondeterministic. A multi-process architecture also ensures that access to persistent state, network, and other nondeterministic external resources is virtualized, making it much easier to make these resources behave in a deterministic manner. Without a multi-process architecture, many of these invasive abstractions (strong interfaces, non-shared state, virtualized resources) must be reimplemented as prerequisites for deterministic replay.

3.2 Design

The remainder of this chapter describes the design and implementation of Dolos, a deterministic replay infrastructure for web programs. The primary purpose of Dolos is to enhance existing workflows (Chapter 5) and enable new developer tools (Chapter 6) by making it possible to revisit past program states within a single execution.

Concretely, the design of Dolos supports these use cases with following requirements:

1. **Low overhead.** Recording must introduce minimal performance overhead, because many web programs are performance-sensitive. Replaying must be fast so that users can quickly revisit past program states.

2. **Exact re-execution.** Recordings must exactly reproduce observable web program behavior when replaying. Replaying should not have effects on the network, persistent state, or other external resources.

3. **Non-interference.** Deterministic replay must not interfere with the use of tools such as breakpoints, profilers, element inspectors, and logging. (Source-to-source instrumentation in particular is disallowed by this requirement.)

4. **Deployability.** It should be possible to casually use deterministic replay functionality without special hardware, installation, configuration, or elevated user privileges.

These requirements induce significant design constraints that have not been fully addressed by prior work (further discussed in Section 2.1.1). The closest points in the design space of deterministic replay techniques are those developed for operating systems [10] and virtual machines [40]. Like these systems, Dolos provides an execution environment on which arbitrary programs can be captured and replayed without modifications. Dolos must mediate access to nondeterministic resources and APIs while allowing nondeterministic and deterministic programs to execute side-by-side.

3.2.1 Types of Nondeterminism

Dolos achieves low overhead by effectively *virtualizing* sources of nondeterminism and otherwise executing a web program using the rendering engine's normal code paths. From the rendering engine's point of view, the web program just so happens to make the same requests every time; from the web program's point of view, the rendering engine just so happens to behave the same way every time.

Compared to other execution environments [57, 91, 138], web program executions are easier to capture and replay in some aspects, and more difficult in others. The single-threaded execution model, cooperative scheduling, and memory safety of web programs make it unnecessary to record thread schedules, instruction counts, and low-level hardware/register states. On the other hand, the plethora of high-level client APIs, low-level platform APIs, modes of interaction, and complexities of retrofitting a virtual machine make it very difficult to find and address all sources of nondeterminism that affect execution.

Nondeterminism manifests in two ways as a web program executes. Table 3.1 provides an overview of sources of nondeterminism that are common to all web rendering engines. *Environmental inputs* are values returned by nondeterministic browser APIs as they are called by JavaScript code. Web programs use these APIs to detect device characteristics, access persistent storage, and interact with external resources. *Event loop inputs* are nondeterministic events that drive execution by evaluating new code, dispatching DOM events, or running JavaScript callbacks directly. Event loop tasks are received by the rendering engine, enqueued into the main event loop, and later executed. Most event loop inputs originate from outside of the rendering engine, and consume an entire event loop turn.

Nondeterminism originates from both internal and external sources [10]. In rendering engines, internal nondeterminism arises when the rendering engine itself needs to schedule event loop tasks; if the tasks can transitively cause JavaScript to execute, then the contents and ordering of these tasks with respect to other event loop inputs is a source of nondeterminism. Section 3.3.3 discusses some examples of internal nondeterminism encountered in the WebKit rendering engine.

External nondeterminism originates from outside of the rendering engine; in a multiprocess browser architecture, these correspond to messages sent between the rendering process and other processes. External nondeterminism can manifest as both event loop inputs and environmental inputs. An example of the latter is most forms of user input: when a user types characters, their browser sends the rendering engine several keyboard events which are enqueued into the rendering engine's event loop and later handled. An example of the former are application-specific policies and persistent states. When a user clicks on a link, the rendering engine first asks the browser whether the proposed navigation is allowed by the browser's security policy before proceeding. Data storage for persistent state (cookies, local storage, etc.) is managed by the browser and accessed as needed by the rendering engine.

Input	Classification	DOM Events & APIs
Keyboard strokes	Event Loop	`keyup`, `keypress`, `keydown`
Mouse input	Event Loop	`mouseover`, `click`
Scroll wheel	Event Loop	`scroll`, `mousewheel`
Page focus/blur	Event Loop	`focus`, `blur`
Window resize	Event Loop	`resize`
Document navigation	Event Loop	`unload`, `pagehide`
Timer callbacks	Event Loop	`setTimeout`, `Promise`
Asynchronous events	Event Loop	animation events
Network response	Event Loop	AJAX, images, data
Random numbers	Environment	`Math.random`
Browser properties	Environment	`window.navigator`
Current time	Environment	`Date.now`
Resource cache	Environment	(none)
Persistent state	Environment	`document.cookie`
Policy decisions	Environment	`beforeunload`

Table 3.1: Major sources of external nondeterminism in rendering engines.

3.2.2 Intercession Mechanisms

Dolos uses three mechanisms to mediate sources of nondeterminism during capturing and replaying. The *capture/inject* mechanism intercepts event loop inputs when capturing an execution, and later injects the captured inputs during re-execution. The *save/restore* mechanism takes a snapshot of an initial state when capturing, and restores the state snapshot when replaying. *Memoization* works at the function call level to save and reuse the results of each invocation. Below, I describe how each of these is deployed to control common sources of nondeterminism in rendering engines.

Dolos uses capture/inject mechanisms exclusively to control event loop inputs. Their task is twofold: to ensure the same computations are enqueued and processed by the rendering engine's event loop on capture and replay; and to prevent any "live" event loop inputs from being processed during playback. For example, if a user started typing as a web program is being replayed, the web program should not process the user's keyboard events because they may diverge execution. Capture/inject mechanisms can intercept and inject event loop inputs either when they are enqueued or as they are processed. Dolos takes the latter approach for reasons described in Section 3.3.1.

Dolos uses both save/restore and memoization mechanisms to control environmental inputs. Save/restore can be used in cases where an initial state—such as a random number seed—can be cheaply and completely saved when capturing begins and restored when playback begins. Once the initial state is restored, subsequent calls to nondeterministic APIs based on this initial state do not need to be handled because execution is assumed to be deterministic. Memoization can also be used to control the same sources of nondeterminism by saving and reusing the values returned every time a related nondeterministic API—such as `Math.random()`—is invoked by JavaScript code. In the case of random numbers, saving the random seed will always require saving less data with the same deterministic effect. In other cases such as per-program persistent database stores, the memoization approach requires less space if the size of the initial state is large and the size of memoized values is small. In many cases, memoization is more straightforward to implement in existing rendering engines if there is not an existing mechanism for restoring an initial state.

3.2.3 Recording and Input Structure

Dolos recordings contain the data necessary to cause a deterministic execution. This data is organized hierarchically in a way that mirrors the structure of execution. At the top level, a recording session consists of one or more segments that correspond to a single

web program execution, as defined in Section 3.1.1. Each segment is self-contained such that the execution it represents can be rearranged, added, or removed from a recording without affecting determinism of other segments[2]. At the next level of hierarchy, each segment contains a sequence of event loop inputs. The first elements of a segment typically represents the initial main frame navigation, actions to save/restore initial state, and then other event loop inputs as observed during capturing..

TODO Make a figure.

3.3 Implementation

Dolos instantiates the deterministic replay strategies outlined above in the context of WebKit [141], a popular rendering engine and browser toolkit. WebKit was chosen because contains the most widely-deployed rendering engine (WebCore) and web developer tools (Web Inspector). Dolos consists of modifications to WebKit's C++ and JavaScript codebase, and can be used as a drop-in rendering engine replacement for Safari by adjusting the dynamic library load path. This section describes implementation details that are specific to Dolos and WebKit. Many of the strategies may apply to other browsers and replay systems, but are not essential to the deterministic replay strategy outlined above. These details have been refined through multiple prototypes which are described in Appendix A.

3.3.1 Capturing and Replaying Executions

Dolos' approach for capturing event loop inputs is to interpose on them just before they are executed, rather than as they are enqueued. Correspondingly, during playback Dolos recreates the actions of event loop inputs by re-executing (instead of enqueuing) them in order. In effect, Dolos captures the subset of the work performed in the rendering engine's

[2]This segmentation scheme is also useful as a crude checkpointing mechanism: since a segment does not depend on the execution of earlier segments, playback can begin from any segment. Checkpointing is further discussed in Section 3.4.4.

event loop that can possibly run JavaScript code or impact the program's determinism. During playback, that same subset of work is initated by Dolos without going through the event loop. The main benefit of this scheme is that executing event loop inputs is synchronous and only spans a single event loop turn. This makes certain functionality—such as pausing playback, aborting playback, or detecting unexpected execution—much easier to implement.

For any given event loop input, there may be multiple levels at which one could capture and inject the event. For example, when a user clicks the mouse, a series of steps occur: first, a `handleMousePress` message is sent to rendering engine from the parent process; second, the user input event is hit-tested against the view hierarchy to find potential event targets; third, the user input is translated into multiple DOM events such as `mousedown`, `click`, `dblclick`, or `dragstart`, depending on the target element and prior inputs; finally, each DOM event is dispatched to elements within the DOM tree, which may cause registered JavaScript event handlers to execute.

Unlike other some prior work that explores deterministic replay, Dolos attempts to capture event loop input events as early as possible in the processing pipeline outlined above. Mugshot [88] and other tools that rely on source instrumentation [98, 112] typically capture later in the pipeline before[3] or while inputs are dispatched as DOM events. Capturing DOM events is more difficult, as it requires serializing event data and the element in the DOM tree to which the element was dispatched. This approach also has lower execution fidelity: rendering engines can perform arbitrary actions prior to dispatching related DOM events, such as moving form field focus, interacting with an input method editor (IME), or other state changes. By contrast, the strategy Dolos uses requires minimal memory use, runtime overhead, and implementation complexity because it copies simple, uninterpreted data structures before they are processed by the rendering engine pipeline. There is no need for Dolos to serialize the event target's position within the

document tree because hit testing is deterministic.

3.3.2 External Nondeterminism

Dolos captures user input events and navigation events as they are sent to the rendering process using a capture/inject mechanism. Each message is saved to the active recording segment when capturing; during playback, each message is redelivered through the same code paths as the original message. "Live" messages during playback, such as a user typing, are not delivered to avoid divergence. WebKit uses platform-provided event loop implementations to handle incoming messages from multiple processes. Thus, the exact interleavings that Dolos records are determined in part by the underlying event loop implementation, and are generally not deterministic.

Dolos also captures network traffic in much the same way as user input events. In WebKit, external resources—such as HTML, JavaScript, and images—are loaded asynchronously and in parallel. As each resource is downloaded, the networking process sends status update message to the rendering process. The rendering engine can use these messages to generate placeholders, dispatch DOM events, or parse and run new JavaScript code. When capturing, Dolos saves these status update messages, their HTTP headers, and associated raw data. When replaying, Dolos blocks outbound network requests from actually reaching the network process, and redelivers status update messages. For example, when loading images asynchronously on Flickr, Dolos saves images as opaque byte buffers split across multiple "data recieved" status update messages. When replaying, Dolos redelivers status update messages with saved image data, and never communicates with Flickr servers.

The DOM provides several mechanisms that allow web programs to schedule asynchronous work, such as `window.setTimeout()`, `window.requestAnimationFrame()`, and the `Promise` API. Similar to other event loop inputs described above, Dolos captures and injects these

[3]Guo et al. [50] report on the space and time benefits of memoizing application-level API calls instead of low-level system calls [119].

callbacks as they are executed. During playback, Dolos manually executes callbacks in the observed order and prevents new callbacks from being scheduled during playback.

Environmental inputs are handled through a combination of memoization and save/restore mechanisms, according to the implementation complexity and space requirements of each method. At the time of writing, most DOM APIs, such as `window.navigator`, `window.screenX`, `window.localStorage` and `window.cookie`, are handled by adding memoization to the code that marshals data between JavaScript and C++. Cookies are an instructive case for whether to memoize or save/restore nondeterministic values. Most calls to `window.cookie` return the same value every time, suggesting that memoization wastes space. However, at the time of writing, cookie storage is handled by the browser instead of the rendering engine, so saving and restoring cookie state from within the rendering engine would require significant refactoring. Since repeated values in a recording can be interned (in-memory) or compressed (serialized), space is not the primary concern.

For environmental inputs handled through memoization, Dolos intercepts calls from the rendering engine to the underlying nondeterministic platform or port APIs instead of memoizing calls to nondeterministic JavaScript functions. For example, Dolos does not record the return value of JavaScript's `Date.now()` function; instead, Dolos memoizes the JavaScript engine's calls to the `currentTimeMS()` platform API inside its implementation of the `Date.now()` function.

3.3.3 Internal Nondeterminism

Because Dolos ensures full determinism of JavaScript, it must take an expansive, pessimistic view of what constitutes nondeterminism. Since simply executing JavaScript can cause divergence, Dolos must mediate all code paths that directly execute JavaScript or indirectly dispatch DOM events (and thus giving JavaScript code a chance to execute). WebKit's rendering engine often uses single-shot timers to enforce asynchronous dispatching of DOM events. For example, `load`, and `error` DOM events are dispatched asynchronously

after an image or stylesheet is successfully parsed. Internally, WebKit uses asynchronous timers to defer the event dispatch. Dolos handles these internal asynchronous mechanisms in the same way that it controls public APIs for scheduling asynchronous work. TODO Give another example, or point to mechanics reference page.

3.4 Evaluation

To the best of our knowledge, Dolos is the first deterministic replay infrastructure integrated directly into a rendering engine. Is replaying this way actually faster? Is it more work to implement? What are its fundametal and incidental limitations? This section characterizes the Dolos approach to replay in terms of performance, replay fidelity, and scalability. This section focuses on the technical aspects of replay; later chapters demonstrate how Dolos can be extended (**??**) and used as the basis for interactive debugging tools (Chapter 5).

3.4.1 Fidelity

The Dolos infrastructure attempts to reproduce[4] identical, deterministic JavaScript executions when capturing and replaying. There is no guarantee regarding number of layout reflows or paints due to time compression or internal rendering engine nondeterminism. This is unimportant because visual output cannot affect the determinism of JavaScript computation. It is possible for JavaScript code to synchronously query the results of computed layout. Methods that perform such queries, such as `Element.offsetLeft`, implicitly suspend all other parsing and Javascript execution until layout results have been updated. Thus, several layout runs may be coalesced, but results will always appear deterministic from the point of view of JavaScript code. Painting and compositing are not observable from client-side JavaScript, and thus are not addressed.

[4]Chapter **??** describes several runtime mechanisms that can detect when and where execution has diverged.

3.4.2 Performance

In our experience, Dolos scales well to real-world interactive web programs without significantly impacting the user experience. Dolos only saves nondeterministic inputs when capturing an execution, so the recordings are very small and can be easily transferred and replayed on other computers. This section provides a historical snapshot[5] of Dolos' performance and maintainability characteristics. Subsequent prototypes (Appendix A) use the same architecture for capturing and replaying, but may be somewhat faster because the WebKit rendering engine continues to recieve general performance optimizations.

Table 3.2 describes performance characteristics of Dolos in a variety of modes and sample web programs. All numbers report the geometric mean of 10 runs (except for interactive runs, which were recorded once but replayed 10 times). The standard deviation was always less than 10% of the geometric mean. Dolos cleared network resource caches between executions to avoid memory and disk cache nondeterminism. In measurements of execution time, local copies of benchmarks were used to avoid nondeterminism caused by network latency and contention. Recording overhead and the amount of data collected scales with user events, network responses, and uses of environmental inputs, not CPU time. Due to replay's negligible performance impact, little effort has been spent thus far to optimize the replay infrastructure.

Recording has almost no time overhead: execution times are dominated by the subject program. Dolos's record/replay performance slowdown is unnoticeable ($< 1.1\times$) for interactive workloads and modest ($\leq 1.65\times$) for non-interactive benchmarks without any significant optimization efforts. Replaying at $1\times$ speed is marginally slower than a normal execution due to extra work performed by Dolos, and seeking (fast replaying) is much faster because it elides user and network waits from the recorded execution.

While being created or replayed, recordings are stored in-memory and consume modest amounts of memory (first column in the data size section of Table 3.2). When seri-

[5] All measurements and numbers in this section were obtained in Spring 2013 using the `timelapse-git` prototype, as described in Section A.1.3.

alized, the recordings are highly compressible. A recording's length is limited only by main memory; in user studies (Chapter 7), users attempted to minimize recording length to reduce the number of inputs that must be later searched.

3.4.3 Scalability of the Approach

Dolos's design scales to new platforms, ports, and sources of nondeterminism, and represents a tiny addition to the rendering engine's codebase. The Dolos infrastructure consists of 7.6K SLOC (74 new files, 75 modified files). For comparison, WebKit contains about 1.38M SLOC. Intersession mechanisms are typically installed at existing module boundaries, such as within the cross-language bindings between DOM and JavaScript, or at the rendering engine's process boundaries. Implementing Dolos required minimal changes to WebKit's architecture. Cases where clear boundaries already existed—such as user inputs, network traffic, and random numbers—made it easy to deploy intercession mechanisms. More substantial efforts were required in cases that lacked these boundaries, such as splitting execution into segments and handling internal nondeterminism.

3.4.4 Limitations

Dolos only ensures deterministic execution for *client-side* portions of web applications. It records and simulates client interactions with a remote server, but does not directly capture server-side state. Tools that link client- and server-side execution traces [145] may benefit from the additional runtime context provided by a Dolos recording.

TODO Reference to future work when discussing intentional divergence

TODO Cannot checkpoint the DOM.

Dolos cannot control the determinism of local, external software components such as Flash, Silverlight, or other plugins. However, plugins interact with the browser and rendering engine via well-defined APIs; Dolos could capture and reproduce these interactions (within the scope of the rendering engine) using capture/inject and memoization

mechanisms.

The Dolos prototype does not address all known sources of nondeterminism, such as the Touch, Battery, Sensor, Screen, or Clipboard APIs, among others. There are no conceptual barriers to supporting these features: they are implemented in terms of standardized DOM events and interfaces, making them relatively easy to interpose upon using mechanisms described in Section 3.2.2. Each new program input requires local changes to route control flow through a mechanisms and new code to marshall the input's data. Event loop inputs additionally require code to inject the input event during playback.

Rendering engines provide many APIs that are can be called nondeterministically by browsers, but do not affect web program determinism. For example, WebKit's rendering engine includes APIs for usability features like native spell-checking, in-page search, and accessibility. Dolos does not attempt to control these nondeterministic APIs because these features do not affect the determinism of the web program, and a developer may wish to use such features differently during playback.

Dolos's hypervisor-like record/replay strategy relies on having good places to virtualize sources of nondeterminism. Rendering engines or execution environments without clear boundaries will require greater engineering efforts to support deterministic replay. For example, the Gecko rendering engine (used by the Firefox browser) is architected as dozens of decoupled components whose instances are shared between multiple web pages. Gecko is also a primarily single-process rendering engine. This design makes it easy to extend the browser and rendering engine, but difficult to record and replay a specific web program in isolation from other web programs or reuse existing interfaces.

3.5 Summary

The Dolos infrastructure is a new approach for exactly reproducing past program states within a single captured execution. It makes the following contributions:

1. A software architecture for integrating deterministic replay into modern browser

rendering engines.

2. An enumeration and categorization of the sources of nondeterminism that impact how a web program executes.

3. Case studies illustrating how three mechanisms can efficiently interpose on important classes of nondeterministic inputs.

4. Evidence that deterministic replay has negligible space and time overheads for interactive web programs.

Program			Run Time						Data Size (KB)	
Name	**Description**	**Workload**	**Bottleneck**	**Baseline**	**Disabled**	**Recording**	**Replaying**	**Seeking**	**Log**	**Si**
JSLinux	x86 emulator	Run until login	network	10.5s	1.00×	1.65×	1.65×	0.37×	24.7 / 71.4 / 8.5	45
JS Raytracer	ray-tracer	Complete run	CPU	6.3s	1.00×	1.01×	1.17×	1.02×	24.0 / 69.4 / 10.2	5
Space Invaders	video game	Scripted gameplay	timers	25.8s	1.00×	1.03×	1.22×	0.25×	247 / 683 / 56.8	7
Mozilla.org	home page	Read latest news	user	22.3s	1.00×	1.00×	1.09×	0.23×	187 / 502 / 29.5	28
CodeMirror	text editor	Edit a document	user	16.6s	1.00×	1.00×	1.03×	0.07×	57.6 / 163 / 17.1	1
Colorpicker	jQuery widget	Reproduce defect	user	15.3s	1.00×	1.00×	1.07×	0.13×	112 / 302 / 26.7	5
DuckDuckGo	search engine	Browse results	user	14.1s	1.00×	1.00×	1.08×	0.19×	119 / 309 / 29.6	19

Table 3.2: Overhead for three non-interactive and four interactive programs. “Baseline” is unmodified WebKit, and “Disabled” is Dolos when neither record nor replay is enabled. Log size is given for the in-memory representation, the uncompressed log file, and the compressed log file. Site content is images, scripts, and HTML.

Chapter 4

LOGGING AND NAVIGATING TO PAST PROGRAM STATES[1]

Reproducing and inspecting specific program states is a fundamental task during activities such as debugging and reverse-engineering. This task is challenging for experts and novices because tools like breakpoints and logging only indirectly support it. When properly placed, logging statements can alert a programmer to relevant runtime states, but logging statements cannot suspend the program for further inspection. Conversely, breakpoint debuggers can suspend a program and control its execution at a fine-grained level, but can only suspend control flow at *future* times rather than at past program outputs. Because of these limitations, breakpoints and logging are subject to a large gulf of execution, making it difficult to predict whether any particular tool will be helpful for the task at hand.

Prior work has investigated deterministic replay techniques—which operate by controlling sources of nondeterminism at runtime—as a basis for automatically reproducing specific executions. Dolos and similar replay infrastructures [88] TODO addref to related work section only provide affordances for navigating a captured execution by its user inputs, event loop tasks, or low-level signals. To suspend execution at a specific program point, a developer must isolate and replay to the preceding input and then use a breakpoint debugger to drive execution to a specific statement.

In prior work, researchers have observed [22, 66, 115] that developers greatly prefer to navigate executions by *outputs*, rather than by inputs. A program's outputs can correspond to relevant program states, and developers often work backwards from outputs

[1]The results in this chapter appear in part in Burg et al. [23]. Some features were upstreamed to WebKit and shipped in Safari 8; see Section A.1.5 for details.

when attempting to understand runtime behavior [65]. To this end, we contribute two extensions to previous replay systems that realize output-oriented navigation: *time-indexed outputs* and *data probes*.

Time-indexed outputs empower a developer to see the logged output they want to investigate and, with a single click, jump to the exact program statement that produced the output. The algorithm for seeking to time-indexed outputs is simple and incurs little overhead. By reducing the task of reproducing program states to only require a single click, time-indexed outputs make it possible for a developer to easily navigate between task-relevant instants of execution without disruptive context switches.

Our second contribution is *data probes*, a feature that allows a developer to retroactively add logging statements to a captured execution without editing program text and without re-executing the program. A data probe may have multiple *probe expressions* that are evaluated and logged to produce new time-indexed outputs. Like a breakpoint, a probe is placed at a single statement in the program; when the statement executes, the probe's expressions are evaluated to create *probe samples*. Data probes and probe samples are saved across multiple playbacks of a captured execution. Using data probes, a programmer can interactively discover, compare, and navigate to interesting program states in the past or the future without excessive planning or manual effort.

EDIT The rest of this paper explains these two contributions.

4.1 An Example

Time-indexed outputs and data probes are designed to automate the tedious, error-prone tasks of suspending execution and logging specific program states within a captured recording. This section uses a program maintenance task to demonstrate advantageous uses of probes and time-indexed outputs.

Color Picker[2] is a jQuery plugin that implements a color picker widget for RGB and HSV colorspaces. Karla, a developer, uses the widget in her web application. She is investigating a bug that manifests when a user manipulates the color picker's color com-

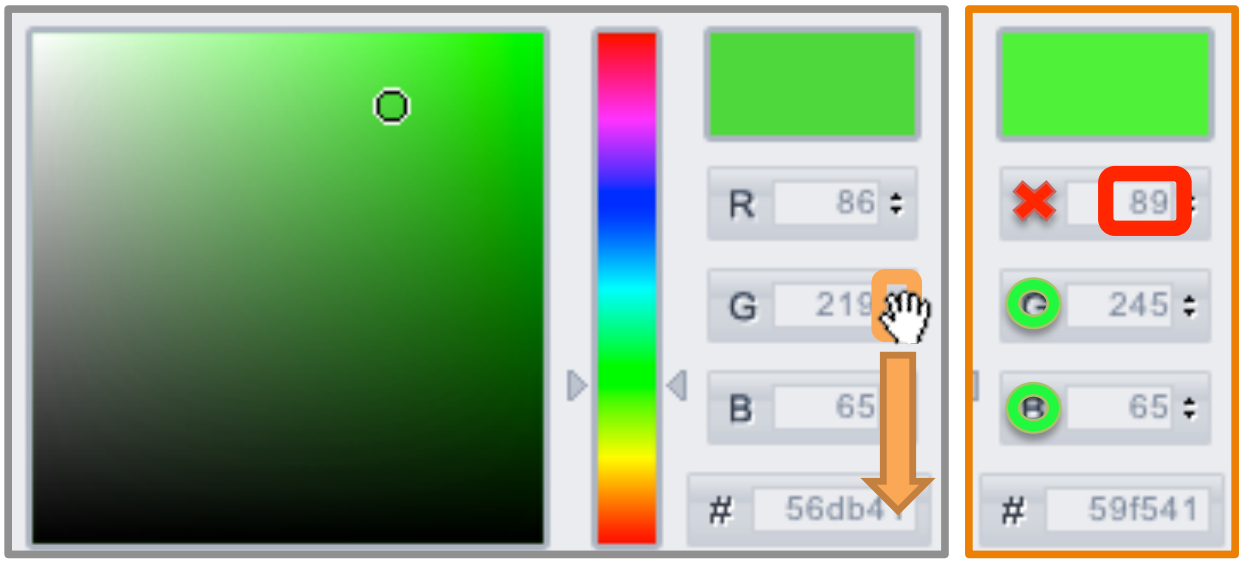

Figure 4.1: The Color Picker widget. When the G color component is adjusted downward (left), the R component unexpectedly changes (right).

ponent sliders (Figure 4.1). Each slider should adjust the value of one red, green, or blue (RGB) component independently of the other two components, but sometimes moving one slider incorrectly affects more than one component. The bug is caused by unnecessary rounding in the algorithm that converts values between RGB and HSV color spaces. Understanding and fixing this bug is difficult for several reasons: reproducing the bug requires manual user interaction; the wrong results appear only sporadically and are not persistent; and it is hard to isolate and investigate specific computations, such as a single event handler execution.

Karla starts by using a record/replay tool (such as Dolos) to capture a recording that demonstrates the steps to reproduce the Color Picker bug. This recording makes further reproduction simple by allowing Karla to quickly jump to the input just prior to the failure, but she still must explore the thousands of lines of code that execute after this input

[2]Colorpicker: **`http://www.eyecon.ro/colorpicker/`**

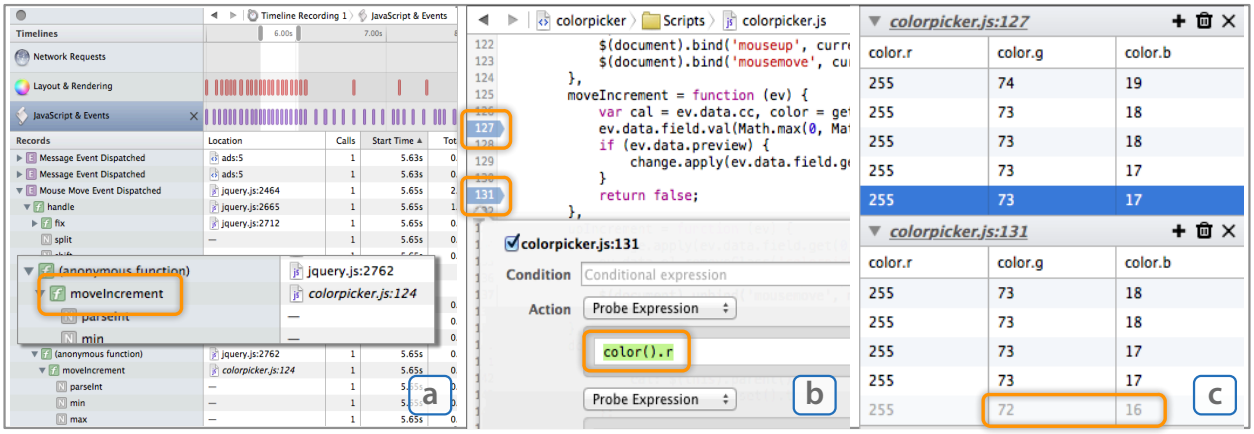

Figure 4.2: Using data probes to revert execution to relevant program states. In (a), Karla inspects the captured recording to find the `moveIncrement` drag event handler (highlighted). In (b), she adds two *data probes* (at lines 127 and 131) to log how color component values changed. In (c), she reverts execution directly to a probe sample (top, selected row) that immediately precedes erroneous component values (below, highlighted).

using traditional debugging tools. She still must set breakpoints to suspend execution at specific lines of code, which is tedious because the program must be re-executed to test whether the breakpoints were positioned correctly. Logging program states is similarly laborious: to log color component changes, Karla has to edit the widget's source code, rebuild and re-deploy the website, capture a new recording, and then view the logged outputs.

Data probes and time-indexed outputs simplify Karla's investigation of the buggy interaction. To create a foothold for observing runtime behavior, Karla uses data probes to log RGB values as they change, since past component values are not stored or logged to the console. After using the built-in timelines view (Figure 4.2.a) to see what code executed during the recording, she guesses that the `moveIncrement` handler may contain the RGB values she wants to log. To see the effects of each drag event, Karla adds data probes before and after the handler modifies color components (at `colorpicker.js:127`

and `colorpicker.js:131`, as seen in Figure 4.2.b). These data probes capture the value of the expressions `color.r`, `color.g`, and `color.b` (Figure 4.2.c) each time the associated line executes. Karla then replays the recording again to generate new probe samples. As the recording is replayed, the probe sidebar (Figure 4.2.c) begins to populate with probe samples. Looking at temporally-ordered probe samples in the console (Figure 4.3), Karla quickly sees a few instances where multiple components changed in a single drag event.

To better understand what happened, Karla wants to inspect the program states leading up to a suspicious probe sample. Since all probe samples are also time-indexed outputs, Karla can suspend execution immediately before the offending drag event handler by double-clicking on a probe sample collected at that time (Figure 4.2.c and Figure 4.3). From that instant of execution, she can use the debugger to step into the event handler and work towards the root cause with the aid of actual runtime values. With time-indexed outputs and data probes, she's likely able to do this much faster and more systematically than with breakpoints and logging alone.

4.2 Implementation

Our prototype of data probes and time-indexed output is built on top of the Dolos infrastructure. Dolos, data probes, and time-indexed outputs have been incorporated into the open-source WebKit browser toolkit[3].

4.2.1 Creating and Evaluating Data Probes

The goal of data probes is to add temporary logging expressions to a program as it is running. Like breakpoints, probes are associated with a specific source statement and are processed just prior to the statement's execution. In fact, our prototype implements probes as a special breakpoint action that evaluates the probe's expressions and saves the results. Probe expressions can capture a wide range of values, including scalars such as

[3] **`https://www.webkit.org`**

numbers or strings, or non-scalar values, such as arrays, objects, or in the case of the web, DOM elements.

The probes user interface supports comparing, relating, and navigating to probe samples. The probes sidebar (Figure 4.2.c) persists collected samples for all subsequent playbacks of the recording. This allows a user to stitch together a map of probe samples across the whole recording without necessarily replaying it contiguously with a specific set of data probes. The probes sidebar groups probe samples by call site to support comparisons. Probe samples are also printed to the console in execution order to provide a navigable, time-synchronized log. During playback, the console shows output produced up to the current instant, but not later outputs captured in a previous playback.

4.2.2 Replaying to Output-Producing Statements

The goal of time-indexed outputs is to suspend execution at the instant when a statement produces a specific output.[4] In a deterministic record/replay setting, output-producing statement evaluations can be uniquely indexed by associating a counter with each such statement. The counter increments when its statement executes, and the produced output is tagged with the current counter value. To suspend execution at the statement that produced time-indexed output n, a naive approach is to set a breakpoint at the statement, re-execute, and resume from the breakpoint $n - 1$ times. However, this approach is too slow and brittle for realistic use: since counter values are relative to the beginning of the recording, a full replay is required to suspend execution at output-producing statements or tag probe samples from a new data probe.

We avoid these drawbacks by making counter values relative to the currently executing event loop task. In a deterministic record/replay setting, event loop tasks that cause JavaScript to execute—such as timer callbacks, network callbacks, or user inputs—are always executed in the same order. If a replay infrastructure can uniquely refer to specific

[4]We assume that tools can automatically identify output-producing statements. Our prototype tags outputs from data probes and `console.log` statements.

event loop tasks, then an evaluation of an output-producing statement can be uniquely indexed by the statement's counter value and the preceding event loop task. This optimization has several important implications:

- We can revisit time-indexed outputs without restarting playback from the beginning of a recording.

- We accumulate partial knowledge of time-indexed outputs across multiple non-contiguous playbacks.

- We can automatically discover new probe samples (i.e., time-indexed outputs) in unknown sections of a recording.

Our prototype adds counter values by modifying the implementations of `console.log` and data probes to tag outputs that these statements produce. The prototype seeks to a specific time-indexed output in three phases: first, it uses Dolos to replay up to the preceding event loop task; second, a hidden breakpoint is added at the output-producing statement; third, the debugger pauses and resumes $n - 1$ times from the hidden breakpoint; and last, execution is suspended a final time immediately before the desired evaluation.

4.2.3 Minimizing Breakpoint Use

While breakpoints are a useful mechanism for collecting probe samples and replaying to time-indexed outputs, their use incurs a significant ($10\times$) performance overhead.[5] This slowdown can negate the interactive qualities of data probes and time-indexed outputs, which may lead a developer to use them in a more cumbersome batch-oriented manner. Our prototype uses several strategies to minimize the use of breakpoints. When replaying to a time-indexed output, it completely disables the debugger until the preceding event loop task, and then sets a single breakpoint at the output-producing statement. This limits

breakpoint-induced slowdowns to a single event loop turn. Our prototype also optimizes its use of breakpoints when sampling data probes. To preserve the developer tool's ability to interactively inspect complex JavaScript values, data probes must be resampled on subsequent playbacks. However, if a probe sample can be serialized and reused without a live object reference, then resampling can be avoided.

4.3 Related Work

Several lines of research investigate ways of tightening the feedback loop of editing code, looking at output, and debugging further. This section discusses those which use program output as beacons for navigating through an execution, and those which simplify the process of gathering runtime state.

4.3.1 Capturing and navigating executions

Two approaches—deterministic record/replay and post-mortem trace analysis—dominate research into capturing and navigating through executions. Deterministic replay research traditionally focuses on capturing executions with low overhead and high fidelity. Recent prototypes such as Aftersight [28] and Jalangi [122] perform dynamic analysis on-demand during replayed executions. Data probes also gather data from a replayed execution, but their placement is driven by user interaction rather than pre-defined analyses.

Tools based on *post-mortem trace analyses* save all program operations into a large trace file, and later query the trace to reconstruct intermediate program states or program output. These systems incur 1–2 orders of magnitude slowdown during recording and amortize that cost by never re-executing the program. In practice, only deterministic record/replay tools have low enough overheads to be used for interactive programs.

The Whyline for Java [65] is a heavyweight trace analysis tool that allows developers to ask why- and why-not questions about program output. It can reconstruct a program's

[5]The mere presence of breakpoints deoptimizes emitted bytecode and prevents most adaptive optimizations such as inline caches.

visual output, textual output, and intermediate program state from the operation trace, and it uses program slicing techniques over the trace to answer program understanding questions on demand. By leveraging deterministic replay instead of post-mortem trace analyses, time-indexed outputs provides many of the same affordances as Whyline, but with near-zero runtime overhead and at interactive speeds on real web applications.

4.3.2 Live programming systems

The live programming paradigm [52] emphasizes tight feedback loops, typically by blurring or removing delays between editing a program and seeing effects of the changes. Live programming tools support quick iteration on inputs or the program itself, and often eschew imperative programming models to better support these goals. Below, we discuss two analogues to data probes and time-indexed outputs designed for other programming environments. While the affordances offered are similar, only data probes and time-indexed outputs are designed for the mainstream domain of imperative, interactive programs such as web applications.

DejaVu [61] supports interactive debugging and development of image processing algorithms. Similar to data probes, a user can add new debugging outputs (e.g. image filters, inferred skeletal models) which are automatically computed and juxtaposed with prior output on a timeline. Using the output timeline, the user can revert execution to a specific input or rendered output frame and inspect the program's state. Time-indexed outputs support a similar interaction for textual outputs, such as console logging or probe samples.

YingYang [86] is a programming environment that integrates live execution feedback with an emphasis on *traces* (similar to console output) and *probes* (similar to our probes). It supports rewinding execution to a logged output, and can substitute concrete values into the code/stack frame to explain the output's derivation. YingYang is built atop the Glitch live programming runtime. Glitch incrementally repairs program state as inputs

or the program itself changes, so it has no notion of state over time, nor does it support temporally-ordered inputs. In contrast, our data probes and time-indexed output can be used to navigate recordings of imperative, interactive programs.

4.4 Future Work

Data probes and time-indexed outputs are a step towards the Whyline [65] vision of interactively investigating runtime behaviors. Data probes and time-indexed output provide a way to navigate captured recordings via console outputs and intermediate script states. In future work, we will continue to explore the Whyline vision while maintaining interactive performance and integration with production web development tools. We want to improve links between visual output and the code fragments that produced the output. We also plan to expand the scope of probes to include other program states, such as changes to the DOM tree structure or changes to an element's appearance or position.

```
[colorpicker.js:127] color.r 255 color.g 49 color.b 13
[colorpicker.js:131] color.r 255 color.g 49 color.b 13
[colorpicker.js:127] color.r 255 color.g 49 color.b 13
[colorpicker.js:131] color.r 255 color.g 50 color.b 14
[colorpicker.js:127] color.r 255 color.g 50 color.b 14
[colorpicker.js:131] color.r 255 color.g 51 color.b 15
```

Figure 4.3: Probe samples ordered temporally in console output. The selected row (green) shows both G and B components changing at the same time. Karla double-clicks on the preceding probe sample to suspend execution prior to the faulty event handler's execution.

Chapter 5

AN INTERFACE FOR CAPTURING AND NAVIGATING EXECUTIONS[1]

EDIT Though it seems that the capability to record and replay executions should be useful for debugging, no prior work has described actual use cases for these capabilities, or how to best expose these capabilities via user interface affordances. At most, prior systems demonstrate feasibility by providing a simple VCR-like interface [88, 138] that can record and replay linearly. Many record/replay systems have no UI, and are controlled via commands to the debugger or special APIs [50, 62, 119].

EDIT This paper presents Timelapse, a developer tool for capturing and replaying web application behaviors during debugging.

EDIT A developer can use Timelapse's interactive visualizations of program inputs to find and seek non-linearly through the recording, and then use the debugger or other tools to understand program behavior.

We discuss the design of Timelapse, and use a scenario to illustrate how Timelapse supports recording, reproducing, and debugging interactive behaviors. Next,

5.1 Reproducing and Navigating Program States

The Timelapse developer tool is designed around two activities: capturing user-demonstrated program behaviors, and quickly and repeatedly reaching program states within a recording when localizing and understanding a fault. The novel features we describe in this section support these activities by making it simple to record program behavior and by providing visualizations and affordances that quicken the process of finding and seeking

[1]The results in this chapter appear in part in Burg et al. [22].

to relevant program states without manually reproducing behavior.

To better understand the utility of replay capabilities during debugging, we first conducted a small pilot study with a prototype record/replay interface. Using contextual inquiry, we found that developers primarily used the prototype to isolate buggy output, and to quickly reach specific states when working backwards from buggy output towards its root cause. Towards these ends, we saw several common use cases: "play and watch"; isolating output using random-access seeking; stepping through execution in single-input increments, and reading low-level input details or logged output.

This section introduces the novel features of the Timelapse developer tool by showing how a fictional developer named Steph might use Timelapse's features while debugging.

TODO There's no discussion of the actual user interface, what encodings it uses/used, etc. Need to put this somewhere in the chapter.

TODO The walkthrough example should be more self-contained, it kind of overtakes the entire chapter. Make better use of "For example," to tie functionality back to the example.

5.1.1 Example: (Buggy) Space Invaders

Steph, a new hire at a game company, has been asked to fix a bug in a web application version of the Space Invaders video game[2]. In this game, the player moves a defending ship and shoots advancing aliens. The game's implementation is representative of modern object-oriented interactive web programs: it uses timer callbacks, event-driven programming, and helper libraries. The game contains a defect that allows multiple player bullets to be in flight at a time; there is only supposed to be one player bullet at a time (Figure 5.1).

5.1.2 Reproducing Program Behavior

Steph is unfamiliar with the Space Invaders implementation, so her first step towards understanding and fixing the multiple-bullet defect is to figure out how to reliably re-

[2]Space Invaders: **`http://matthaynes.net/playground/javascript/glow/spaceinvaders/`**

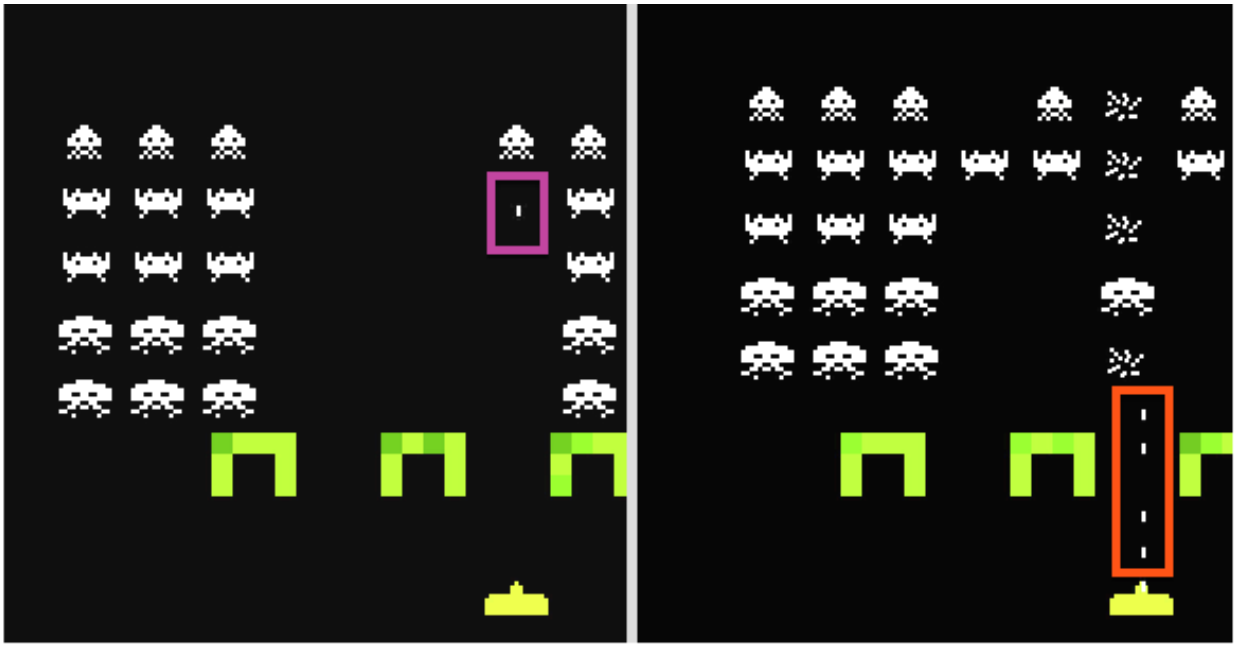

Figure 5.1: Screenshots of normal and buggy Space Invader game mechanics. Only one bullet should be in play at a time (shown on left). Due to misuse of library code, each bullet fires two asynchronous `bulletDied` events instead of one event when it is destroyed. The double dispatch sometimes enables multiple player bullets being in play at once (shown on right). This happens when two bullets are created: one between two `bulletDied` events, and the other after both events.

produce it. This is difficult because the failure only occurs in specific game states and is influenced by execution conditions outside of her control, such as random numbers, the current time, or asynchronous network requests.

With Timelapse, Steph begins capturing program behaviors with a single click (Figure 5.2.6), plays the game until she reproduces the failure, and then finishes the recording. Recordings created by Timelapse are compact, self-contained, and serializable, so Steph can attach her recording to a bug report or share it via email.

To reproduce the defect with traditional tools, Steph would have to multitask between

synthesizing reproduction steps, playing the video game, and reflecting on whether the reproduction steps are correct. Once Steph finds reliable reproduction steps, she could then use breakpoints to further understand the defect. But, breakpoints might themselves affect timing, making the defect harder to trigger or requiring modified reproduction steps.

5.1.3 Navigating to Specific Program States

To focus her attention on code relevant to the failure, Steph needs to know which specific user input—and thus which event handler invocations—caused the second bullet to appear.

Steph uses Timelapse's visualization and navigation tools (Figure 5.2) to locate and seek the recording to an instance of the multiple bullets failure. First, Steph limits the zoom interval to when she actually fired bullets, and then filters out non-keystroke inputs. She replays single keystrokes with a keyboard shortcut until a second bullet is added to the game board. Then, she seeks execution backward by one keystroke. At this point, she is confident that the code which created the second bullet ran in response to the current keystroke.

Without Timelapse, it would not be possible for Steph to isolate the failure to a specific keystroke and then work backwards from the keystroke. Instead, she would have to insert logging statements, repeatedly reproduce the failure to generate logging output, and scrutinize logged values for clues leading towards the root cause.

5.1.4 Navigation Aid: Debugger Bookmarks

Having tracked down the second bullet to a specific user input, Steph now needs to investigate what code ran, and why.

With Timelapse, Steph sets several *debugger bookmarks* (Figure 5.3.6) at positions in the recording that she wants to quickly revert back to, such as the keystroke that caused

the second bullet to appear or an important program point reached via the debugger. Debugger bookmarks support the concept of temporal focus points [60, 123], which are useful when a developer wants to relate information [66]—such as the program's state at a breakpoint hit—that is only available at certain points of execution. Timelapse restores a debugger bookmark by seeking to the preceding input, setting and continuing to the preceding breakpoint, and finally simulating the user's sequence of debugger commands (step forward/into/out).

With traditional tools, Steph must explore an execution with debugger commands such as "step into", "step over", and "step out". This is frustrating because these commands are irreversible, and Steph would have to manually reproduce the failure multiple times to compare multiple instants or the effects of different commands.

5.1.5 Navigation Aid: Breakpoint Radar

Once Steph finds the code that creates bullets, she needs to understand why some keystrokes fire bullets and others do not.

With Timelapse, Steph first adjusts the zoom interval to include keystrokes that did and did not trigger the failure. Then, she sets a breakpoint inside the `Bullet.create()` method and records and visualizes when it is actually hit during the execution using the *breakpoint radar* feature (Figure 5.3.3a). Breakpoint radar automates the process of replaying the execution, pausing and resuming at each hit, and visualizing when the debugger paused. Steph can easily see which keystrokes created bullets and which did not.

With traditional tools, Steph would need to repeatedly set and unset breakpoints in order to determine which keystrokes did or *did not* create bullets. To populate the breakpoint radar timeline, Steph would have to manually hit and continue from dozens or hundreds of breakpoints, and collect and visualize breakpoint hits herself. For this particular defect, breakpoints interfere with the timing of the bullet's frame-based animations, so it would be nearly impossible for Steph to pause execution when two bullets are in

flight.

TODO How does this relate to probes?

5.1.6 *Interacting with Other Debugging Tools*

Once Steph localizes the part of the program responsible for the multiple bullets, she still needs to isolate and fix the root cause. To do so, Steph uses debugging strategies that do not require Timelapse, but nonetheless benefit from it. Timelapse is designed to be used alongside other debugging tools such as breakpoints[3], logging, and element inspectors; its interface can be juxtaposed (Figure 5.3) with other tools.

Through code inspection, Steph observes that the creation of a bullet is guarded by a flag indicating whether any bullets are already on the game board. The flag is set inside the `Bullet.create()` method and cleared inside the `Bullet.die()` method. To test her intuition about the code's behavior, she inserts logging code and captures a new execution to see if the method calls are balanced. The logging output in Figure 5.4 is synchronized with the replay position: as Steph seeks execution forward or backward, logging output up to the current instant is displayed. Logging output is cleared when a fresh execution begins (i.e., Timelapse seeks backwards) and then populated as the program executes normally.

Steph has discovered that the multiple-bullet defect is caused by the `bulletDied` event being fired twice, allowing a second replacement bullet to be created if the bullet "fire" key is pressed between the two event dispatches. In other words, the failure is triggered by firing a bullet while another bullet is being destroyed (by collision or leaving the game board).

With basic record/replay functionality, affordances for navigating the stream of recorded inputs, and the ability to easily reach program states by jumping directly to breakpoint hits, Timelapse both eliminates the need for Steph to repeatedly reproduce the Space Invaders failure and frees her to focus on understanding the program's logic. EDIT Our user

[3]To prevent breakpoints from interfering with tool use cases, Timelapse tweaks breakpoints in several ways: breakpoints are disabled when recording or seeking, and enabled during real-time playback.

evaluation explores these benefits further.

5.2 Discussion

5.3 Conclusion

TODO Recap on whether this was a good UI, what changed, etc.

Figure 5.2: The Timelapse tool interface presents multiple linked views of recorded program inputs. Above, the timelines drawer (1) is juxtaposed with a detailed view of program inputs (2). The recording overview (3) shows inputs over time with a stacked line graph, colored by category. The overview's selected region is displayed in greater detail in the heatmap view (4). Circle sizes indicate input density per category. In each view, the red cursor (5) indicates the current replay position and can be dragged. Buttons allow for recording (6), 1× speed replay (7), and breakpoint scanning (8). Details for the selected circle (9a) are shown in a side panel (9b).

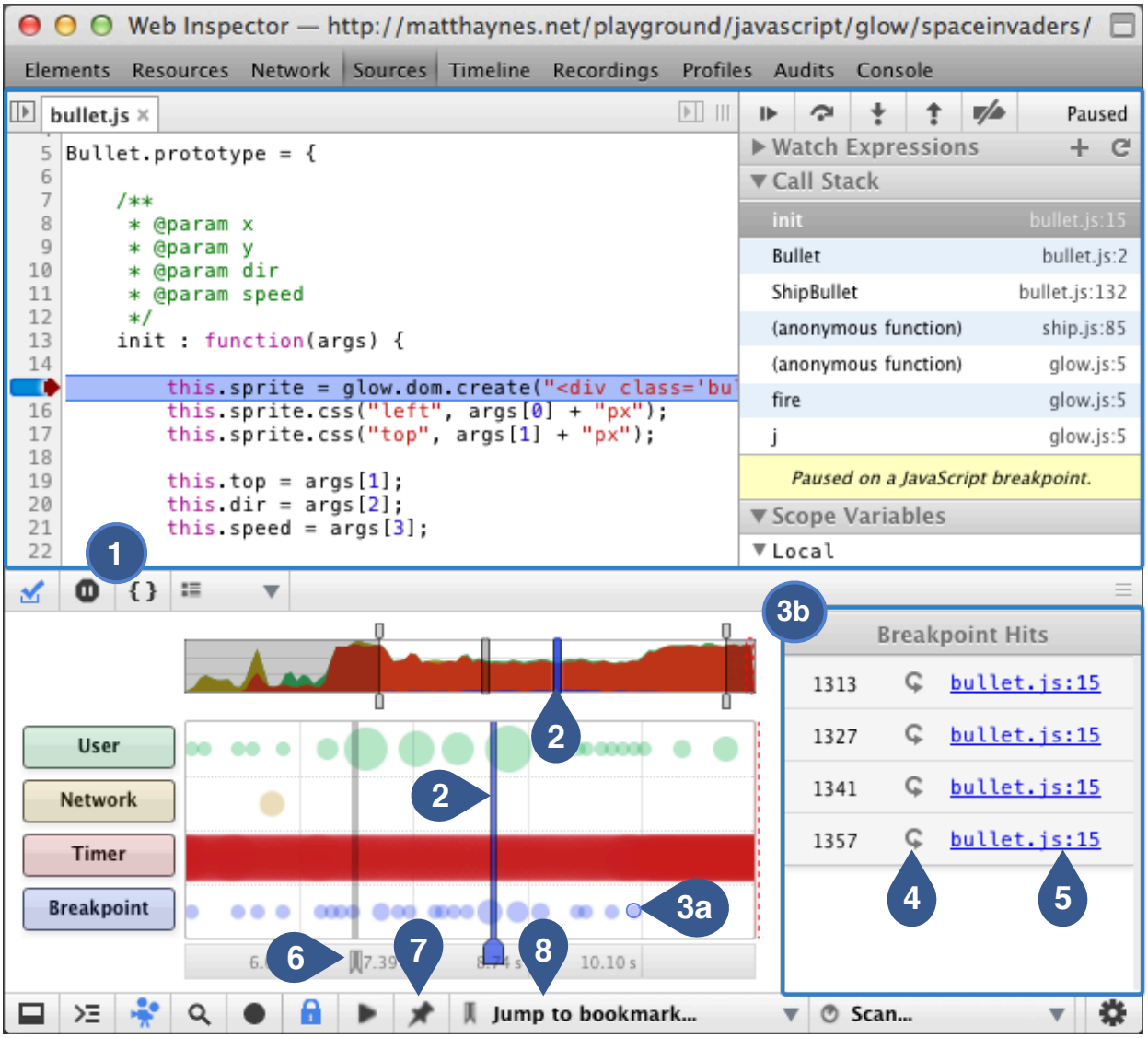

Figure 5.3: Timelapse's visualization of debugger status and breakpoint history, juxtaposed with the existing Sources panel and debugger (1). A blue cursor (2) indicates that replay execution is paused at a breakpoint, instead of between user inputs (as shown in Figure 5.2). Blue circles mark the location of known breakpoint hits, and are added or removed automatically as breakpoints change. A side panel (3b) shows the selected (3a) circle's breakpoints. Shortcuts allow for jumping to a specific breakpoint hit (4) or source location (5). Debugger bookmarks (6) are set with a button (7) and replayed to by clicking (6) or by using a drop-down menu (8).

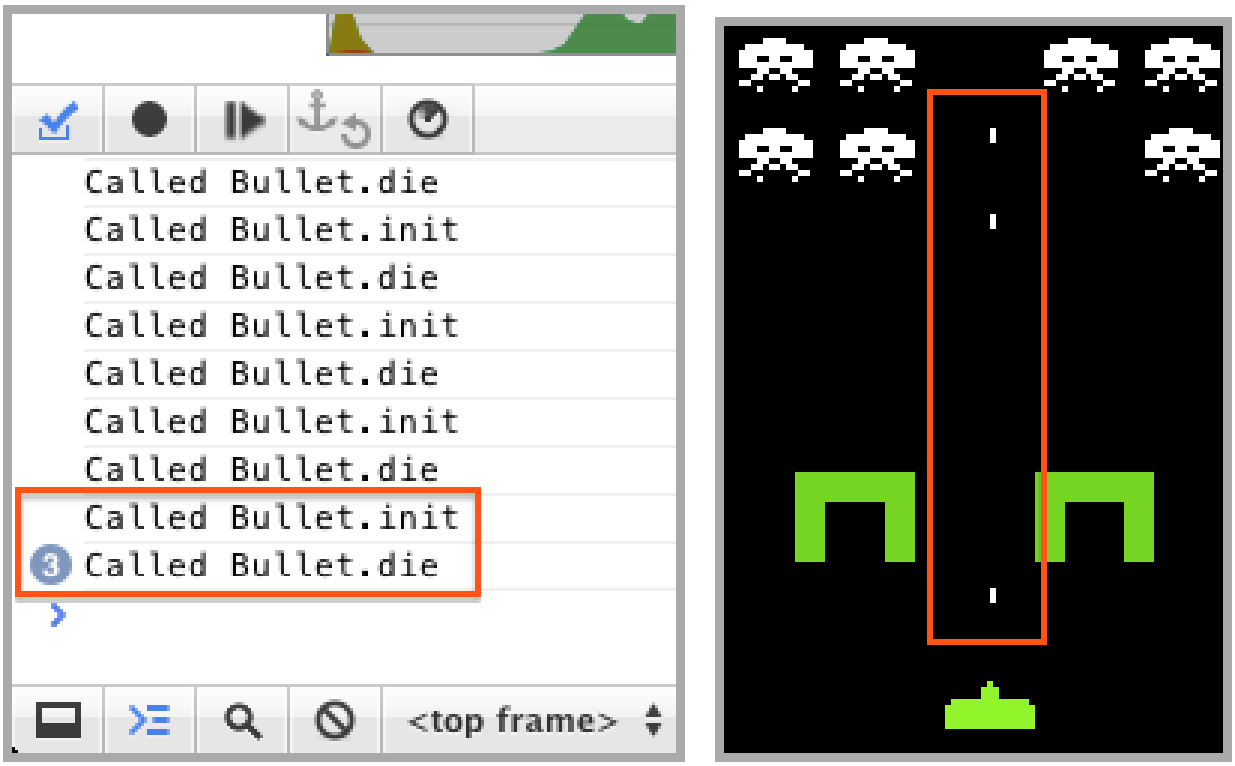

Figure 5.4: Screenshots of the logging output window and the defect's manifestation in the game. Steph added logging statements to the `die` and `init` methods. At left, the logging output shows the order of method entries, with the blue circle summarizing 3 identical logging outputs. According to the last 4 logging statements (outlined in red), calls to `die` and `init` are unbalanced. At right, three in-flight bullets correspond to the three calls to `Bullet.init`.

Chapter 6

EXPLAINING VISUAL CHANGES IN WEB INTERFACES[1]

Web developers increasingly look to existing web site designs for inspiration [41], to learn about new practices or APIs [19], and to copy and adapt interactive behaviors for their own purposes [127]. Web sites are particularly conducive to reuse because web pages are widely available, distributed in source form, and inspectable using tools built into web browsers.

Unfortunately, when a developer finds an interactive behavior that they want to reuse (e.g., a nicely designed widget, a parallax effect, or a slick new scrolling animation), finding the code that implements the behavior on a third-party web site is a challenging process [68]. Locating this code typically involves at least three tasks. First, a developer identifies a behavior's rendered *outputs* and speculates about the internal states that produce them; second, she observes changes to these outputs over time to find their *internal states*; third, she searches the web site's client side implementation to find the JavaScript *source code* that implements the behavior.

Even if all of these tasks go well, extracting the implementation of the desired behavior might require repeated experimentation and inspection of the running site. More often, the web's combination of declarative CSS, imperative JavaScript, and notoriously complex layout/rendering algorithms makes the feature location task prohibitively difficult. Moreover, as the web sites and web applications become more powerful, they have also become more complex, more obfuscated, and more abstract, aggravating these challenges.

While prior work has investigated feature location techniques for statically-typed, non-dynamic languages, no prior work comprehensively addresses the specific difficul-

[1]The results in this chapter appear in part in Burg et al. [24].

ties posed by the web. (1) Isolating the output, internal state and source code for single widget on a web page is difficult due to hidden and non-local interactions between the DOM, imperative JavaScript code, and declarative CSS styles. Some prior work has addressed this by revealing *all* hidden interactions [98], but this obfuscates critical example-relevant details in a flood of low-level information. (2) Web developers can view program states and outputs over time by repurposing tools for logging, profiling, testing, or deterministic replay [5, 22, 88], but none of these tools is designed to compare past states, and they cannot limit data collection to specific interface elements or behaviors. (3) No prior work exists that can attribute changes in web page states to specific lines of JavaScript code. Instead, web developers often resort to using breakpoints, logging, or browsing program text in the hope of finding relevant code [22, 77]. In other programming environments, research prototypes with this functionality all incur run-time overheads that limit their use to post-mortem debugging [55, 65, 95].

TODO Minor editing to match chapter style. Overall this chapter is in good shape. To address these gaps, we present Scry, a reverse-engineering tool that enables a web developer to (1) identify visual states in a live execution, (2) browse and compare relevant program states, and (3) jump directly from state differences to the JavaScript code responsible for the change. (1) To locate an interactive behavior's implementation using Scry, a developer first identifies a web page element to track. Whenever the selected interface element is redrawn differently, Scry automatically captures a snapshot of the element's visual appearance and all relevant internal state used to render it. (2) Scry presents an interactive diff interface to show the CSS and DOM differences that caused any two visual states to differ, overlaying inline annotations to compactly summarize CSS and DOM changes between two visual states. (3) When a developer clicks an annotation, Scry reveals the operations that caused the output change and the corresponding JavaScript code that performed the operation. Scry supports these capabilities by capturing state snapshots, logging a trace of relevant mutation operations, and tracking dependencies between operations. The result is a powerful, direct-manipulation, before-and-after approach to feature location for

the web, eliminating the need for developers to speculate about and search for relevant code.

This chapter begins with an illustration of how Scry helps a developer as they locate the code that implements a mosaic widget. Then, it explains the design rationale and features of Scry's user interface, and it describes Scry's snapshotting, comparison, and dependency tracking techniques. EDIT It concludes with several real-world case studies, related work, and future directions.

6.1 Example: Understanding a Mosaic Widget

To illustrate Scry in use, consider Steph, a contract web developer who is overhauling a non-profit organization's website to be more engaging and interactive. While browsing another website[2], she finds a compelling picture mosaic widget (Figure 6.1) that might work well on the non-profit's donors page. To evaluate the widget's technical suitability, she needs a high-level understanding of how it is implemented in terms of DOM and CSS manipulations and the underlying JavaScript code. In particular, she wants to know more about the widget's cross-fade animation: its dependencies on specific DOM and CSS features, its configurability, and ease of maintenance. At this point, Steph is only superficially familiar with the example: how it looks visually and a vague intuition for what it does operationally. She is unfamiliar with the example's source code, and she does not desire a complete understanding of it unless absolutely necessary.

Existing developer tools provide several approaches for Steph to reverse-engineer the mosaic widget to gain this understanding, but all of these are ill-suited for her task. She could search through the page's thousands of lines of source code for functions and event handlers relevant to the mosaic and then try to comprehend them. Steph is unlikely to pursue this option as it is extremely time-consuming, and it might not aid her evaluation. She could inspect the page's output to see its related DOM tree elements and active CSS

[2] `https://www.mozilla.org/en-US/mission/`

Figure 6.1: A picture mosaic widget that periodically switches image tiles with a cross-fade animation. It is a jQuery widget implemented in 975 lines of uncommented, minified JavaScript across 4 files. Its output is produced using the DOM, CSS animations, and asynchronous timers.

rules, but this only shows the page's current state and does not explain how the page's DOM tree or styles were constructed. She could use source-level tools (e.g., execution profiler, logging, breakpoints) to see what code actually executes when the widget animates. However, the efficient use of these tools requires *a priori* awareness of what code is relevant, and Steph is unfamiliar with the example's code.

6.1.1 Finding Code that Implements a Mosaic Widget

Using Scry, Steph can identify the mosaic's relevant visual outputs, compare internal states that produced each output, and jump from internal state changes to the responsible JavaScript code. Instead of guessing about program states and searching static code, Steph's workflow is grounded by output examples, captured DOM and CSS states, and specific lines of JavaScript code. Steph starts by finding the mosaic's corresponding DOM element using the Web Inspector, and then tells Scry to track changes to the element (Figure 6.2.a). As mosaic tiles update, Scry captures snapshots of the mosaic's internal state and visual output. After several tile transitions, Steph stops tracking and browses the collected snapshots to identify visual states before, during, and after a single tile changes images (Figure 6.2.b).

Steph now wants to compare these output examples to see how their internal states differ as the cross-fade effect progresses. To do so, she selects two screenshots from the timeline (Figure 6.2.b). For each selected screenshot, Scry shows the small subset of the page's DOM (Figure 6.2.b) and CSS (Figure 6.2.c) that determines the mosaic's visual appearance at that time. By viewing the specific inputs and outputs of the browser's rendering algorithm at each instant, Steph can figure out how the mosaic widget is structured and laid out using DOM elements and CSS styles. To highlight dynamic behaviors, Scry visualizes differences between the states' DOM trees and CSS styles (Figure 6.2.c). Seeing that the tiles' `background-image` and `opacity` properties have changed, Steph now knows which CSS and DOM properties the mosaic uses to implement the cross-fade.

To find the code that causes these changes, Steph compares the DOM trees of the initial state and mid-transition state, noticing that the new tile initially appears underneath the original tile, and the original tile's `opacity` style property differs between the two states. When she selects the new tile's DOM element, Scry displays a list of JavaScript-initiated mutation operations that created and appended DOM elements for the new tile (Figure 6.2.d). To see how the tile's `opacity` property is animated, she clicks on its diff

annotation and sees JavaScript stack traces for the state mutations that animate the style property (Figure 6.2.e,f). Steph now knows exactly how the widget's JavaScript, DOM, and CSS code works together to animate a tile's cross-fade transition. If Steph wants to modify the animation code, she now knows several places within the code from which to expand her understanding of the program.

6.2 A Staged Interface for Feature Location

User interfaces for searching and understanding code can quickly become overwhelming, displaying large amounts of source code to filter, browse, and comprehend [118]. Scry's interface simplifies this work by identifying and supporting three distinct activities through dedicated interfaces: (1) the user identifies the behavior's major visual states, (2) she builds a mental model of how visual outputs are related to internal states, and (3) she explores how multiple internal states are connected via scripted behaviors. This section first describes and justifies this output- and difference-centric workflow; then, it explains how Scry's design supports a web developer during each of these feature-location activities.

6.2.1 Design Rationale

We designed Scry to directly address the information overload a developer encounters when performing feature location tasks [118]. Scry's design differs from traditional feature location tools in two fundamental ways: (1) Scry represents program states by their visual output whenever possible, and (2) Scry promotes a staged approach to feature location by iteratively showing more detailed information. We explain each of these points below.

Representing Interface States as Screenshots

Scry's user interface removes much of the guesswork from feature location, by using visual outputs as the primary basis for identifying and comparing an interactive behavior's intermediate internal states. Scry shows multiple output examples for an element along with the internal states (CSS styles and DOM elements) used to render each output (Figure 6.3). A user browses internal states by selecting the corresponding screenshots that each internal state produces. This output-based, example-first design is in contrast to the traditional tooling emphasis on static, textual program representations. During feature location tasks, browsing program states via output examples is a better match for what the user knows (a visual memory of a page's output) and what they lack but are seeking (knowledge of relevant state and code). Output examples are also more readily available: visual states are easier for developers to recognize and compare than internal states or static code, and output often changes in response to distinct and memorable user actions.

Performing Feature Location in Stages

Scry's interface allows a developer to pursue specific feature location tasks in relative isolation from each other. When a user wants to identify outputs, relate outputs to internal states, or connect state changes to source code, Scry provides only the information appropriate to each task.

To see the internal states that produced a single visual output, they can do so with without considering scripted behaviors and other interactions. While most interactive behaviors are scripted with JavaScript[3], ultimately an element's visual appearance is solely determined by its CSS styles and a DOM tree. Thus, to see how one visual state is produced, it is sufficient to understand the CSS and DOM that were used to render it. Scry supports this task by juxtaposing each screenshot with its corresponding DOM tree and computed CSS style properties (Figure 6.3).

[3]Simple interactions can be programmed entirely within declarative style rules using CSS animations,

To direct a user towards the internal states (CSS and DOM) responsible for visual changes, Scry's interface visualizes differences between two screenshots and their corresponding DOM and CSS states (Figure 6.4). Sometimes, inspecting the internal state and visual output of single visual state is insufficient for a useful mental model of how internal inputs affected visual output. If the user has a weak understanding of CSS or layout algorithms, or if the interface element is excessively large or complex, then it may be difficult to localize a visual effect to specific CSS styles and DOM elements. By juxtaposing small changes in internal state with the corresponding visual outputs, Scry prompts a user to test and refine their mental model against a small, understandable example. These diffs also reveal the means by which JavaScript code is able to transition between different visual states.

To help a user understand how state differences came to be and what code was responsible, Scry explains how each DOM and CSS difference came to be in terms of abstract *mutation operations* that modify CSS styles or the DOM tree. Each mutation operation serves a dual purpose: it jointly explains how internal state changed, and also provides a starting point from which users can plug in their own search and navigation strategies [77] to find other relevant code upstream from the mutation operation. Initially, the user is presented with a list of recorded operations for one state difference; the user can browse these operations to understand how the state changed. Once the user wants to see the source code responsible for these mutation operations, they click on a specific operation to see where it was performed. Many mutation operations originate from source code (**??**), such as JavaScript function calls or assignments that cause some change to the DOM or CSS. Since these mutations may happen indirectly—for example, by adding a class, setting `node.innerHTML`, or by changing styles from JavaScript—there can be multiple JavaScript statements responsible for a change.

---

transitions, and pseudo-states (i.e, `:hover`, `:focus`) to specify keyframes. Scry can track these internal state changes even though no JavaScript is involved.

6.2.2 *Capturing Changes to Visual Appearance*

Scry automatically tracks changes to a user-specified DOM element's appearance and summarizes the element's output history with a series of screenshots. To start tracking an element, a developer first locates a *target element* of interest, using existing tools such as an element inspector or DOM tree viewer. Once the developer issues Scry's "Start Tracking" command (Figure 6.2.a), Scry immediately begins capturing a log of mutation operations for the entire document. When Scry detects changes in the target element's visual appearance, it captures a state snapshot and adds a screenshot to the target element's tracking timeline. (We later explain how these tracking capabilities are implemented.)

The element tracking timeline (Figure 6.3.a) is Scry's primary interface for viewing and selecting output examples. It juxtaposes these output examples—previewable screenshots of the target element—with existing timelines for familiar run-time events such as network activity, script execution, page layout, and asynchronous tasks. Timelines show events on a linear time scale and can be panned, zoomed, and filtered to focus on specific interactions or event types.

6.2.3 *Relating Output to Internal States*

Scry's snapshot interface enables a developer to learn how an element's visual appearance is rendered by juxtaposing inputs and outputs of the browser's rendering algorithm[4]. After a developer has captured relevant output states of the target element, she then selects a single screenshot from the timeline (Figure 6.3.a) to see more details about that visual state. The visual output, DOM subtree, and computed CSS styles for a single visual state are shown together in the snapshot detail view (Figure 6.3). To help a developer understand how the visual output was rendered, the visual output and CSS views are linked to the DOM tree view's current selection. When a developer selects a DOM element (Fig-

ure 6.3.c), Scry shows the element's matched CSS styles (Figure 6.3.d).

6.2.4 Comparing Internal States

Using Scry's comparison interface (Figure 6.4), a developer can quickly compare internal states of two relevant output examples to learn why the examples were rendered differently. Scry automatically discards CSS styles that are overridden in the rule cascade. Thus, the differences in two snapshots' input data—its CSS styles and DOM tree—are sufficient to explain differences in their output data.

The comparison interface (Figure 6.4) consists of two side-by-side snapshot interface views with additional annotations to indicate the nature of their differences. Additions are annotated with green highlights, and only appear within the temporally-later snapshot. Removals are annotated with red highlights, and only appear within the earlier snapshot. Modifications—a combination of an addition and removal for the same style property or DOM attribute—appear in purple highlights for both snapshots. Elements whose parent has changed are highlighted in yellow, and elements whose matched CSS styles have changed are rendered in bold text. As with the single snapshot view, a developer can inspect a DOM tree element to see its matching CSS styles and position within visual output. In the comparison tool, the view state of both sides is kept in sync so that the element is selected (if present) in both snapshots. This allows the developer to easily compare CSS styles and DOM states without having to recreate the same view for the other snapshot.

6.2.5 Relating State Differences to JavaScript Code

To complete the link from output examples to JavaScript, Scry computes which mutation operations were responsible for producing specific CSS or DOM state differences. To view

[4]Scry does not directly explain causal relationships between inputs and outputs in the style of Whyline [65]. Instead, Scry helps a developer, who has a working understanding of CSS-based layout, by providing concrete data against which they can validate their mental model of layout.

the mutation operations for a difference, a developer selects a colored highlight from the comparison interface (Figure 6.4.a). Then, Scry changes views to show the difference alongside a list of mutation operations (Figure 6.5.b) that caused the difference. Each operation includes a JavaScript stack trace that shows the calling context for the mutation operation (Figure 6.5.c). Using this link, a developer can find pieces of code related to a single visually-significant difference.

6.3 Implementation

Scry's functionality is realized through four core features: detecting changes to an element's visual output; capturing input/output state snapshots; computing fine-grained differences between state snapshots; and capturing and relating mutation operations to state differences.

6.3.1 Detecting Changes to Visual Output

A central component of Scry's implementation is the *state snapshot*, which represents the state of a particular DOM element at a particular point in a program's execution. Before we discuss the data a state snapshot contains and how it is captured, we first discuss how Scry decides when to capture a snapshot of a distinct visual state.

Scry differs from prior work [98] in that it observes actual rendered visual output to detect changes to specific interface elements. When visual output significantly differs, Scry captures and commits a state snapshot. While many input state mutations may occur while JavaScript is running on the page, it is essential to Scry's example-oriented workflow that it only captures states that are visually distinct and are relevant to the target element.

To detect these distinct visual states, Scry intercepts paint notifications from the browser's graphics subsystem and applies image differencing to the rendered output of the DOM element selected by the user. If the painted region does not intersect the selected ele-

ment's bounding box, then Scry knows the element was not updated; if the target element's bounding box differs from its previously observed bounding box, then a snapshot is taken, as its location has moved. To check for output changes, Scry renders the target element's subtree into a separate image buffer and then compares the image data to the most recent screenshot. If the bitmaps have nontrivial differences[5] then Scry takes a full state snapshot of the target element and commits it as a distinct visual state. TODO reformat so this footnote is not a footnote. It keeps getting split across lines.

Rendering and comparing an element's DOM subtree in isolation is surprisingly difficult due to two features of CSS: stacking contexts and transparency. Stacking contexts allow an element's back-to-front layer ordering to be changed by CSS properties such as `opacity`, `transform` or `z-index`. In practice, this can cause ancestor elements to be rendered visually in front of descendant elements and occlude any subtree changes. Scry mitigates this by not rendering ancestor elements and visualizing the target element's bounding box before tracking it. This strategy has shortcomings, however: descendant elements frequently allow ancestor elements to "shine through" transparent regions in order to provide a consistent background color. If ancestors are not rendered, then screenshots will lack the expected background color. To work around this, Scry retains screenshots of the target element with and without ancestor elements if they differ; the background-less version is used to detect visual changes, while the background-included version is shown to the user.

6.3.2 Capturing State Snapshots

When Scry decides to capture a state snapshot, it gathers many details to help a developer understand the state of the selected element in isolation from the rest of the page. Snapshots consist of the screenshot of the element's visual state in bitmap form, the subtree of

[5]To compute image differences, Scry computes the mean pixel-wise intensity difference over the entire bitmap, and uses a threshold of 1% maximum difference. This allows for minor artifacts arising from subpixel text rendering and other nondeterministic rendering behavior.

the DOM rooted at the target element, and the *computed style* for each tree element. Snapshots are fully serialized in order to isolate past visual state snapshots from subsequent mutation operations.

An element's computed style describes the set of properties and values that are ultimately passed forward (but not necessarily used) in the rendering pipeline to influence visual output. In order to trace computed style property values back to specific style rules, inline styles, and mutation operations, Scry performs its own reimplementation of the CSS cascade that tracks the origin of each computed style property. Computed style properties originate from one of four sources: declarative *style rules*, explicit *inline styles*, CSS animations, and inherited properties. In order to later deduce why a style property has changed, Scry saves the CSS rules and specific rule selectors that match each node in the snapshot.

The current Scry implementation does not attempt to capture all of a page's view state (scroll positions, keyboard focus, etc.) or external constraints (window size, locale) in state snapshots. The only exceptions are the CSS pseudo-states `:hover` and `:focus` because they are frequently used by interactive behaviors. If changes to the page's view state cause the target element's appearance or bounding box to change, then Scry will commit a new state snapshot, but it will not have sufficient information to explain how the outputs differ in terms of inputs. Prior work [22] has demonstrated that these view state inputs can be easily and cheaply collected. Insomuch as these inputs affect the set of active CSS rules, they can be treated similarly to inherited style attributes on the target element that may have global effects.

6.3.3 Comparing State Snapshots

Scry's usefulness as a feature location tool hinges on its ability to compute comprehensible state changes between snapshots and relate these to concrete mutation operations and JavaScript code. To precisely compare two snapshots, Scry compares each snapshot's 1)

Input affected	Data affected	JavaScript API / change origin
DOM	Tree Structure	`Node.appendChild`
	Node Attributes	`Node.className`
	Node Content	`Node.textContent`
	Bulk Subtree	`Node.innerHTML`
CSS	Style Rules	*various*
	Inline Styles	`Element.style`
	Animated Properties	`animation` CSS property
	Legacy Attributes	`Element.bgcolor`
View State	Scroll Positions	*user*, `Node.scrollOffset`
	Mouse Hover	*user*
	Keyboard Focus	*user*
Environment	Window Size	*operating system*

Table 6.1: Input mutation operations. View State and Environment are not currently supported in Scry, but are listed for completeness.

captured DOM subtrees and 2) computed styles. In the remainder of this section, we refer to the two snapshots being compared as the *pre-state* and *post-state*.

DOM Trees

Scry compares DOM subtrees and computes change summaries on a per-node basis. To compute an element's change summary, Scry first finds the element in both snapshots. To do this, Scry associates a unique, stable identifier with each DOM node at run time to make it possible to find the same node in two snapshots via a hash table lookup. If Scry finds the corresponding nodes in both snapshots it summarizes differences in their parent-sibling relationships, attributes, and computed styles. A node that appears in only one snapshot is reported as added or removed, and a node whose parent changed or

whose order among siblings changed is reported as moved. This strategy identifies many small, localized changes (Table 6.2) that are straightforward to explain in terms of low-level mutation operations (Table 6.1). Moreover, these summaries correspond to the types of changes that developers are accustomed to reading in text diff interfaces, making them familiar and easy to comprehend.

An alternative strategy for comparing subtrees is to globally summarize changes using tree matching algorithms [68] or tree edit distance algorithms [14]. We found these to be unsuitable for linking small state changes back to JavaScript code. Tree matching algorithms compute per-node similarity metrics, but do not try to attribute per-node dissimilarities to mutation operations. Edit distance algorithms do not directly produce per-node change summaries, and describe mutations using a minimal sequence of abstract tree operations. Web pages' mutation operations do not correspond to tree edit script operations: real edit sequences are often not minimal (for example, repeated mutations of a node's `class` attribute should not be coalesced) and include redundant but useful operations (such as replacing a subtree by assigning to `Node.innerHTML`).

Computed Styles

To compute differences between a single node's computed styles in the pre-state and post-state, Scry uses set operations on CSS property names. To determine which properties were added or removed, it computes the set difference. Property names present in both snapshots are compared to detect whether their property values or origins differ.

6.3.4 Explaining State Differences

When a user selects a specific state difference to see what code was responsible, Scry presents a sequence of JavaScript-initiated mutation operations that caused the difference. Scry computes this causal chain on-demand in three steps. First, using the affected node's change summary, Scry finds a single *direct operation* within its operation log that produces

Input affected	Change type	Cases
DOM	Node Existence	node-added, node-removed
	Relationships	parent-changed, ordinal-changed
	Attributes	attribute-changed, attribute-added, attribute-removed
CSS	Property Existence	property-added, property-removed
	Direct Styles	value-changed
	Indirect Styles	origin-changed

Table 6.2: Possible cases for per-node change summaries produced by comparing state snapshots. Similar cases are shown the same way in the user interface, but are summarized separately to simplify the task of finding a corresponding direct mutation operation.

the node's expected post-state. Second, Scry finds multiple *prerequisite operations* which the direct operation depends on. Lastly, the operations are ordered and presented in the user interface as a causal chain connecting the node's pre-state and post-state.

As a starting point, we first discuss the mutation operations that Scry captures as raw material for producing causal chains. Then, we detail the specific strategies that Scry uses to identify the code responsible for a change: (1) how to identify direct operations for node changes and simple style changes; (2) how to find direct operations that indirectly cause computed styles to change via style rules; (3) and how to compute dependencies between mutation operations.

Capturing Mutation Operations

The web exposes a large, overlapping set of APIs to effect changes to visual appearance by mutating rendering inputs. This section enumerates these input *mutation operations* that Scry must log and relate to state differences. Scry instruments APIs and code paths for

each of the input mutation operations listed in Table 6.1. While tracking a target element, Scry saves a log of these mutation operations for later analysis. At the time that each mutation operation is logged, if the operation is performed by JavaScript code, Scry also captures a call stack, to help the developer link state differences to JavaScript code causing the mutation, and the upstream code and event handlers that caused it to execute.

Mutation operations as defined by Scry (Table 6.1) closely mirror the most commonly used DOM and CSS APIs. These operations can be used to explain changes to DOM state, and changese to computed style properties that originate from style rules (whose rules match and unmatch as the DOM tree changes). Scry also captures mutation operations from other computed style property change origins, such as an element's animations and inline styles set from JavaScript codoe.

Finding Direct Operations for DOM Changes

For a specific state change (Table 6.2), Scry scans backwards through the operation log to find the most recent operation related to the state change. The most recent operation that mutates state into the post-state is the change's direct operation; other prerequisite operations are separately collected as the direct operation's dependencies (described below). For attribute differences, Scry finds the most recent change to the attribute. For tree structure differences, Scry determines what operations could have caused the change and finds the most recent one with the correct operands. For example, if a node's ordinal rank among its siblings differs, then Scry looks for operations that inserted or removed nodes from its parent.

Finding Direct Operations for Style Changes

Scry uses origin-specific strategies to find direct operations for a computed style property change. If a property originates from an inline style that was set from JavaScript, Scry simply scans backwards for a mutation operation that directly assigned that inline

style. If a property originates from a declarative CSS animation or transition, then the browser rendering engine automatically changed the property value, triggered by an element gaining or losing a `animation` property from its computed style. In this case, the user wants to know where the originating `animation` property came from, so Scry finds the direct operation that caused the animation property to change.

If a property originates from a style rule, then Scry must determine which of the element's matched rules changed and relate that to a DOM difference. Properties can be added or removed when rules start or stop matching the element. Changes to a property's value may happen when rules either match or unmatch and change the results of the CSS cascade. Therefore, Scry analyzes how a node's matched rules and selectors differ between snapshots to find what caused different rules to match. To change result of the CSS cascade, either a rule must stop matching and "lose" the property, or a rule must start matching and "win" the property. If the losing rule is not present in the post-state, then Scry looks for state differences between the snapshots that could cause the selector to no longer match. For example, if the rule `div.hidden { display: none; }` stopped matching a `<div>` element, then Scry deduces that a differing `class` attribute caused the rule to stop matching.

Computing Dependencies between Mutation Operations

To provide the user with a sequence of operations that transform the pre-state into the post-state, Scry must compute dependencies between mutation operations. This is similar to the notion of an executable *program slice*: the operation sequence must preserve a specific behavior (cf. a *slicing criterion*), but it is permissible for it to over-approximate and include irrelevant operations. Reducing the operation trace length (cf. slice size) for a state change simply makes it easier for a human to browse and comprehend how the change occurred. Note that these dependencies only ensure that the mutation operations

preserve the specific state changes captured in the pre-state and post-state[6].

Scry computes an operation dependency graph on-demand as a user selects pre-state and post-state snapshots. To produce a causal chain for a change, Scry finds the change's direct operation in the dependency graph, collects operations its transitive closure, and orders operations temporally. Dependencies for operations between the pre-state and post-state are computed in three steps: first, operations are indexed by their node operands. Second, Scry builds a directed acyclic graph with operations as nodes and causal dependencies between operations as directed edges. Operations that do not explicitly depend on other operations implicitly depend on the pre-state. Scry processes operations backwards starting from the post-state; each operation's dependencies are resolved in a depth-first fashion before processing the next most-recent operation. Finally, when all operations have been processed, graph nodes with no outgoing edges (i.e., depend on no other operations) are connected to a node representing the pre-state.

Operations that mutate node attributes and inline styles require the operand nodes and attributes to exist. For example, an attribute-removed operation depends on the existence of a node n and attribute a to remove. If neither n or a existed in the pre-state, then the operation's dependencies include the subsequent mutation operations that created n and/or a. Similarly, operations that change the structure of the DOM (append-child, set-parent, replace-subtree, etc.) require all of their operands to exist.

6.3.5 Prototype Implementation

TODO Add reference to prototypes appendix.

Scry is implemented as a set of modifications to the WebKit browser engine [141] and its Web Inspector developer tool suite. To provide the element tracking user interface, Scry extends the Web Inspector with a new screenshot timeline, snapshot detail and com-

[6]Since JavaScript can access DOM state and layout results, there are untracked control and data dependencies between JavaScript and inputs. We leave dynamic slicing of JavaScript dependencies to future work.

parison views, and integrations between difference summaries and other parts of the interface. Scry also tracks dependencies for mutation operations and finds direct mutation operations in the JavaScript-based frontend. Element screenshots, DOM tree snapshots, style snapshots, and mutation operations are gathered through direct instrumentation of WebKit's WebCore rendering engine and sent to the Web Inspector frontend. Scry tabulates computed styles in C++ with full access to the rendering engine's internal state.

6.4 Practical Experience with Scry

Despite the rise of a few dominant client-side JavaScript programming frameworks, web developers use DOM and CSS in endlessly inventive ways that tool developers cannot fully predict. In our experience, even when Scry's results are diluted by idiosyncratic uses of web features, it is still helpful for at least *some* parts of a feature location task. This section presents several short case studies that illustrate Scry's strengths and weaknesses, motivating future work.

6.4.1 Expanding Search Bar

A National Geographic web article[7] contains a navigation bar with an expanding search field. When the user clicks on the magnifying glass icon, a text field appears and grows to a reasonable size for entering search terms. Without Scry, this behavior is difficult to investigate because the animation lasts less than a second, and intermediate animation states are not displayed or persisted.

To understand this widget, we used the Web Inspector's "Inspect" chooser to locate the search icon element in the DOM tree browser. We then started tracking the element with Scry and interacted with the widget to start its animation. Upon browsing captured screenshots, we saw that the text field's width and opacity both changed. We compared two snapshots with Scry and saw that separate CSS `transition` properties were applied to

[7] National Geographic, *Forest Giant*.
http://webplatform.adobe.com/Demo-for-National-Geographic-Forest-Giant/browser/src/

different tree elements. We clicked on the animated property value and Scry presented a list of mutation operations, revealing that a `click` event handler had added a `.expanded` class to the root element of the widget to trigger an animated transition. In this example, Scry was particularly helpful in two cases: (1) it captured intermediate animated property values which are normally not possible to see in the inspector; and (2) Scry was able to trace the cause of the entire animation back to a single line of JavaScript code that changed an element's class name.

6.4.2 A Tetris Clone

A Tetris-like game[8] uses DOM elements and CSS to render the game's board and interface elements. To understand how the board is implemented using CSS, we used Scry to track changes to the main playing area. As we played the game, Scry took full snapshots of the game board. By inspecting the DOM of each snapshot, it became apparent that the game board is implemented with one container element per row and multiple square-shaped `<div>`s per row to form pieces. When we compared two board states that had no pieces in common, we unexpectedly found that Scry identified two squares on the board as being the same. After following mutation operations into the JavaScript implementation, we discovered that the Tetris game uses an "object pooling" strategy. To produce shapes on the game board using squares, the game reuses a fixed set of DOM elements and explicitly positions them using inline styles. Scry's confusion arose because the game board states happened to reuse the same square elements from the object pool.

From this example, we learned that although Scry's current implementation expects that each allocated DOM element has a consistent identity, many applications violate this expectation. Some client-side rendering frameworks such as React [43] expose an immediate-mode API called the *virtual DOM*. Client JavaScript implements `draw()` methods that fully recreate a widget's DOM tree and CSS styles using the virtual DOM. Behind

[8]Tetris Clone. **`http://timothy.hatcher.name/tetris/`**

the scenes, React synchronizes the virtual and real DOM using a fast tree edit algorithm, reusing the same elements to produce visual output for unrelated model objects that happen to use the same HTML tag names. A similar problem arises with frameworks that re-render a component by filling in an HTML string template and overwriting the component's prior DOM states by setting the target element's `innerHTML` property. In this case, Scry shows the target element's entire subtree as being fully removed and re-added. Scry doesn't try to match similar nodes, but could be extended to fall back to using more relaxed similarity-based metrics [68] instead of strict identity when re-finding DOM elements.

6.4.3 A Fancy Parallax Demo

In the past few years, browsers added support for applying 3D perspective transforms to elements using CSS. The fancy parallax demo[9] discussed here is representative of pages that use scroll events and transforms to implement parallax and infinite scrolling effects. For this page, we wanted to learn how an element's position is computed in response to scrolling events. We used Scry to track an animated paragraph of text as it moved around when we scrolled the page. From a single DOM tree snapshot, we could see that CSS `transform`-related properties were set on all elements subject to scroll-driven animations. We compared two snapshots to find the source of changes to the `transform` properties, and were always led back to the same line of JavaScript code. Looking upstream in the stack trace, it appeared that a JavaScript library interprets the single scroll position change and imperatively updates the `transform` style property for dozens of elements. We could not discover (using Scry) where the animation configurations for each element were specified.

From this example, we learned that Scry is of limited use for localizing code when the endpoints of JavaScript—where it directly interfaces with rendering inputs—are not easily distinguishable. Such *megamorphic* callsites to browser APIs are common when a

[9]Fancy Parallax Demo. **`http://davegamache.com/parallax/`**

web page calls DOM APIs indirectly through utility libraries. A simple solution would be to automatically disambiguate very active callsites based on their calling context, or allow the user to hide library code (known as "script blackboxing" in some browsers). However, for this example, simply filtering the stack traces would not lead a user to the configuration data for a parallax animation. A better solution would be to extend Scry's capabilities to include tracking of control and data dependencies through JavaScript [122]. This would require a very different technical approach, since Scry instruments native browser APIs rather than JavaScript code.

6.5 Discussion and Future Work

Scry is a first step towards demystifying the complex, hidden interactions between the DOM, CSS layout, JavaScript code, and visual output. The capabilities we have described validate our interface concept; other explanatory capabilities could be added without significantly altering Scry's staged, example-oriented workflow. In particular, we see two promising directions for future work: expanding the scope and accuracy of Scry's explanations, and tracking an element's changes backward (rather than only forward) in time.

TODO Consider turning this into a limitations section since it is fairly application-specific. Future work is more about deterministic replay's other applications, not problems with specific UI or tools. The "fix" for the limitations could be a forward or backward reference. For example, tracking causality through JS could link to a related work section on dynamic analysis of JS.

Opening the Layout Black Box Scry treats the layout/rendering pipeline as a black box, but users often want to know how single style properties are used (or not) within the pipeline. Within the design space of black-box approaches, it would be straightforward to extend Scry to further minimize rendering inputs using observation-based slicing [15]. Concretely, Scry could delete individual style properties from a snapshot if the resulting visual output does not differ [144]. Even with a minimal set of inputs, a more involved "white-box" approach to explaining layout [92] would still be extremely useful. By adding more instrumentation to browser rendering engines, Scry could be extended to

directly answer why and why-not questions [64, 65, 92] about how inputs are used or how outputs are derived as they funnel through the increasingly complex layout algorithms of modern web browsers.

Tracking Causality through JavaScript Scry can explain DOM or CSS differences in terms of mutation operations, but it does not track the upstream dependencies in JavaScript code that caused the mutation operations. Scry could be extended with recent work on JavaScript slicing [122] and event modeling [1] to extend its explanations to show an uninterrupted causal chain [65] between user inputs and events, JavaScript state and control flow, mutation operations, and changes to layout inputs and outputs. This would produce explanations of changes that would be both more complete and more precise.

Tracking Past Element States

TODO Replace with a conclusion, and link to future work.

Given a target element, Scry can track its future visual states as a developer demonstrates the behavior of interest by interacting with the page. However, in fault localization tasks a developer often wants to see what went wrong in the past that produced a buggy state in the present. To gather past states of an element, Scry could build on recent deterministic replay frameworks for web applications [22] to collect snapshots and trace data from earlier instants of the execution. Prior work has demonstrated the feasibility of such an "offline dynamic analysis" [28, 29, 122], but none has integrated this technique into a user interface or web browser.

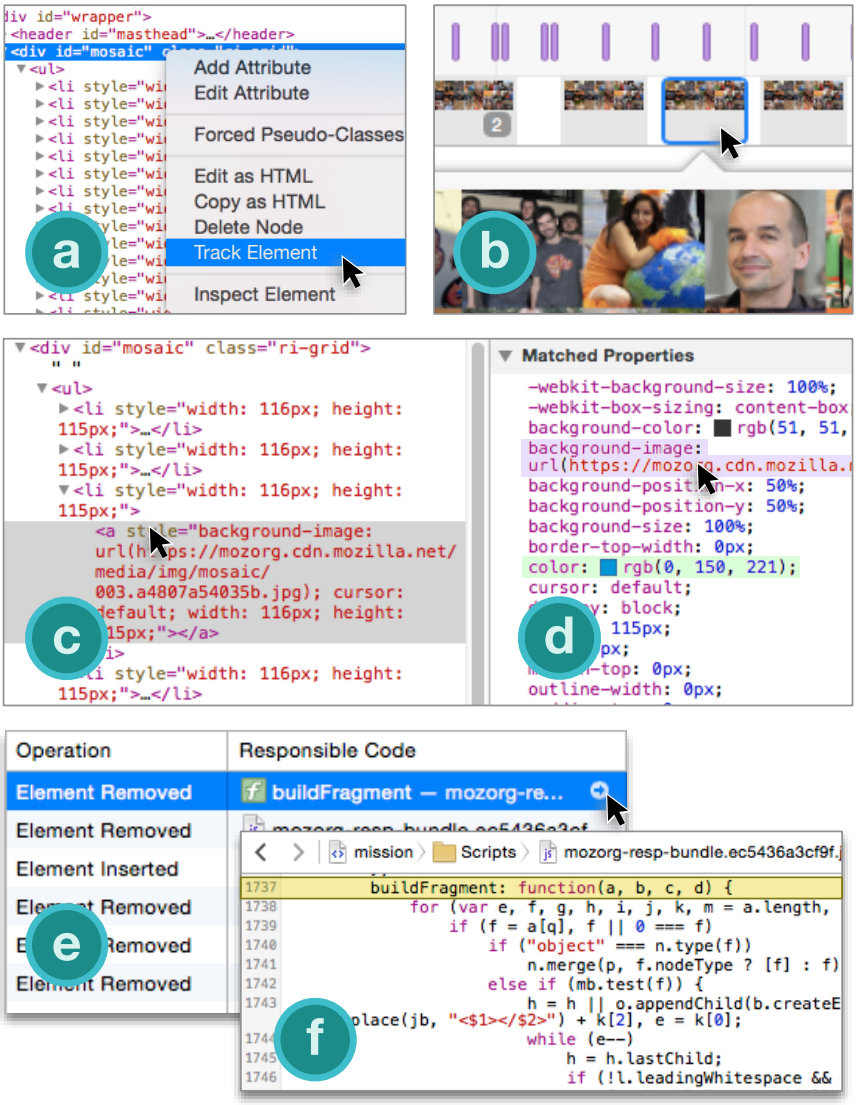

Figure 6.2: Steph first uses the Web Inspector to go from the mosaic's visual output to its DOM elements. Then, she uses Scry to track changes to the mosaic element (a), select different visual states to inspect (b), and see the DOM tree (c) and CSS styles (d) that produced each visual state. To jump to the code that implements interactive behaviors, Steph uses Scry to compare two states and then selects a single style property difference (d). Scry shows the mutation operations responsible for causing the property difference (e), and Steph can jump to JavaScript code (f) that performed each mutation operation.

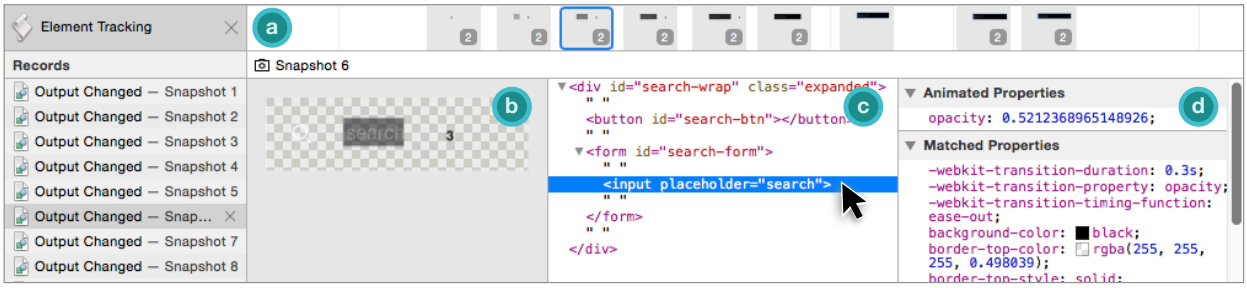

Figure 6.3: Scry's snapshot interface shows multiple screenshots in a timeline view (a). When the developer selects an output example from the timeline, Scry shows three views for it: (b) the element's visual appearance, (c) its corresponding DOM tree, and (d) computed style properties for the selected DOM node.

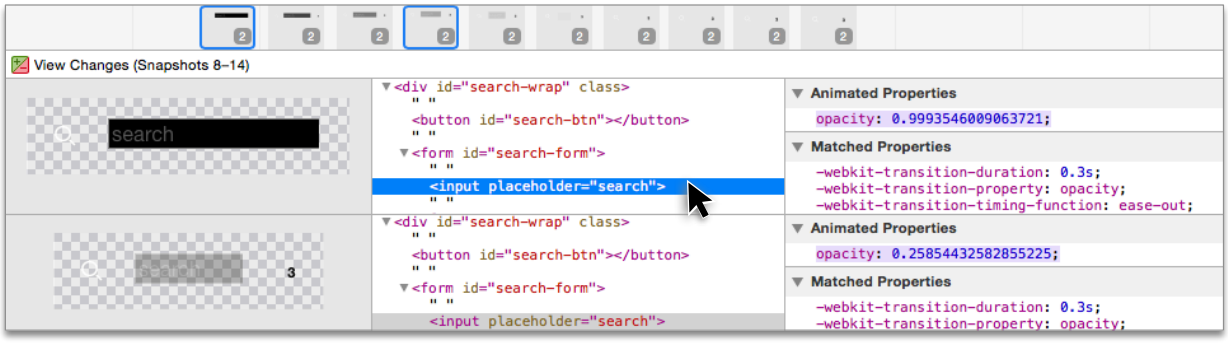

Figure 6.4: The snapshot comparison interface. Two screenshots are selected in the timeline, and their corresponding CSS styles and DOM trees appear below. Differences are highlighted using inline annotations; here, the `opacity` style property has changed. Additions, removals, and changes are highlighted in green, red, and purple, respectively.

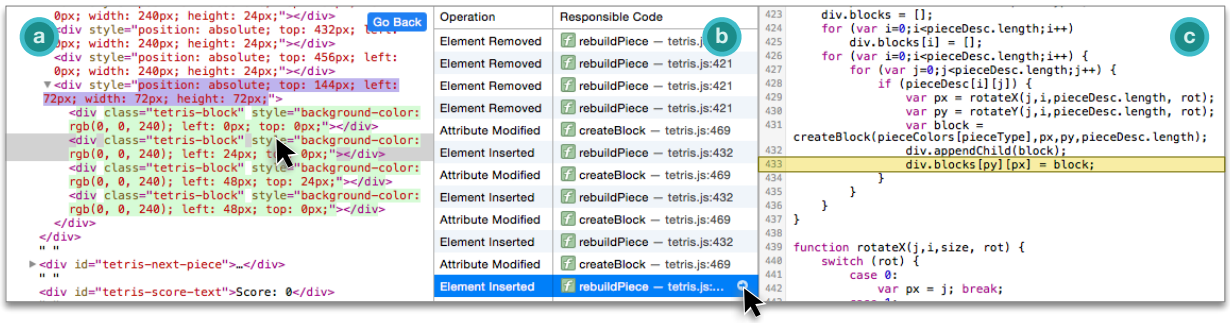

Figure 6.5: Explaining a DOM tree difference in a Tetris game that modifies, removes, and rebuilds DOM elements. The left pane (a) shows a single difference being investigated (the second added `<div>`). The center pane (b) shows a list of mutation operations that caused the change. When an operation is selected, Scry shows a source code preview (c) and call stack (not shown) for the operation.

Chapter 7

HOW DEVELOPERS USE TIMELAPSE[1]

EDIT We discuss the results of a small study to see how developers use Timelapse in open-ended debugging tasks.

Prior work [40, 88, 119] asserts the usefulness of deterministic record and replay for debugging. In this section, we present a formative user study that investigates when, how, and for whom record/replay tools and specifically Timelapse are beneficial. Our specific research questions were:

RQ 1 How does Timelapse affect the way that developers reproduce behavior?

RQ 2 How do successful and unsuccessful developers use Timelapse differently?

7.1 Study Design

We recruited 14 web developers, 2 of which we used in pilot studies to refine the study design. Each participant performed two tasks. We used a within-subjects design to see how Timelapse changed the behavior of individual developers. For one task, participants could use the standard debugging tools included with the Safari web browser. For the other task, they could also use Timelapse. To mitigate learning effects, we randomized the ordering of the two tasks, and randomized task in which they were allowed to use Timelapse.

The goal of this study was to explore the variation in how developers used Timelapse, so the tasks needed to be challenging enough to expose a range of task success. To balance realism with replicability, we chose two tasks of medium difficulty, each with several intermediate milestones. Based on our results, our small exploratory study was still suf-

[1]The results in this chapter appear in part in Burg et al. [22].

ficiently large to capture the variability in debugging and programming skill among web developers.

7.2 Participants

We recruited 14 web developers in the Seattle area. Each participant had recently worked on a substantial interactive website or web application. One half of the participants were developers, designers, or testers. The other half were researchers who wrote web applications in the course of their research. We did not control for experience with the jQuery or Glow libraries used by the programs being debugged.

7.3 Programs and Tasks

7.3.1 Space Invaders

One program was the Space Invaders game from our earlier example scenario. The program consists of 625 SLOC in 6 files (excluding library code) and uses the Glow JavaScript library[2]. We chose this program for two reasons: its extensive use of timers makes it a heavy record/replay workload, and its event-oriented implementation is representative of object-oriented model-view programs, the dominant paradigm for large, interactive web applications.

We asked participants to fix two Space Invaders defects. The first was an API mismatch that occurred when we upgraded the Glow library to a recent version while preparing the program for use in our study. In prior versions, a sprite's coordinates were represented with `x` and `y` properties; in recent versions, coordinates are instead represented with `left` and `top` properties, respectively. After upgrading, the game's hit detection code ceases to work because it references the obsolete property names. The second defect was described in the motivating example and was masked by the first defect.

---

[2]Glow JavaScript Library: **`http://www.bbc.co.uk/glow/`**

7.3.2 *Colorpicker*

The other program was Colorpicker[3], an interactive widget for selecting colors in the RGB and HSV colorspaces (see in Figure 7.1). The program consists of about 500 LOC (excluding library and example code). The widget supports color selection via RGB (red, green, blue) or HSV (hue, saturation, brightness) component values or through several widgets that visualize dimensions of the HSV colorspace.

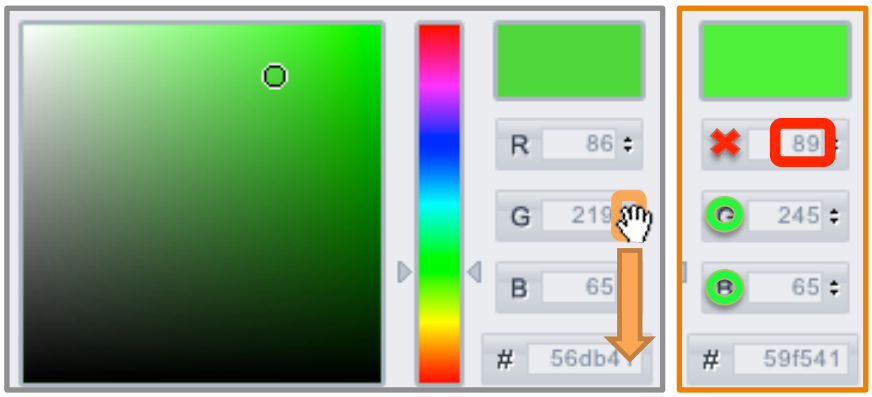

Figure 7.1: The Colorpicker widget.

We chose this program because it makes extensive use of the popular jQuery library, which—by virtue of being highly layered, abstracted, and optimized—makes reasoning about and following the code significantly more laborious.

The Colorpicker task was to create a regression test for a real, unreported defect in the Colorpicker widget. The defect manifests when selecting a color by adjusting an RGB component value, as shown in Figure 7.1. If the user drags the G component (left panel, orange), the R component spontaneously changes (right panel, red). The R component should not change when adjusting the G component. The bug is caused by unnecessary rounding in the algorithm that converts values between RGB and HSV color spaces. Since

[3]Colorpicker widget: **`http://www.eyecon.ro/colorpicker/`**

the color picker uses the HSV representation internally, repeated conversions between RGB and HSV can expose numerical instability during certain patterns of interaction.

We claim that both of these faults are representative of many bugs in interactive programs. Often there is nothing wrong with the user interface or event handling code *per se*, but faults that are buried deep within the application logic are only uncovered by user input or manifest as visual output. The Space Invaders faults lie in incorrect uses of library APIs, but manifest as broken gameplay mechanics. Similarly, the Colorpicker fault exists in a core numerical routine, but is only manifested ephemerally in response to mouse move events.

7.4 Procedure

Participants performed the study alone in a computer lab. Participants were first informed of the general purpose and structure of the study, but not of our research questions to avoid observer and subject expectancy effects. Immediately prior to the task where Timelapse was available, participants spent 30 minutes reading a Timelapse tutorial and performing exercises on a demo program. In order to proceed, participants were required to demonstrate mastery of recording, replaying, zooming, seeking, and using breakpoint radar and debugger bookmarks. Participants could refer back to the tutorial during subsequent tasks.

Each task was described in the form of a bug report that included a brief description of the bug and steps to reproduce the fault. At the start of each task, the participant was instructed to read the entire bug report and then reproduce the fault. Each task was considered complete when the participant demonstrated their correct solution. Participants had up to 45 minutes to complete each task. They were not asked to think aloud[4].

We stopped participants when they had demonstrated successful completion to us or exceeded the time limit.

[4]In an earlier formative study, we solicited design feedback by using a think aloud protocol. We did not do so in the present study to avoid biasing participants' work style.

After both task periods were over, we interviewed participants for 10 minutes about how they used the tool during the tutorial and tool-equipped task and how they might have used the tool on the other task. We also asked about their prior experience in bug reproduction, debugging, and testing activities. Participants who completed the study were compensated for their time.

7.5 Data Collection and Analysis

We captured a screen and audio recording of each participant's session, and gathered timing and occurrence data by reviewing the video recordings after the study concluded.

Our tasks were both realistic and difficult so as to draw out variations in debugging skill and avoid imposing a performance ceiling. We measured task success via completion of several intermediate task steps or critical events. For the Space Invaders task, the steps were: successful fault reproduction, identifying the API mismatch, fixing the API bug, reproducing the rate-of-fire defect, written root cause, and fixing the rate-of-fire defect. For the Colorpicker task, the steps were: successful fault reproduction, written root cause, correct test form, identifying a buggy input, and verifying the test.

We measured the time on task as the duration from the start of the initial reproduction attempt until the task was completed or until the participant ran out of time. We recorded the count and duration of all reproduction activities and whether the activity was mediated by Timelapse (automatic reproduction) or not (manual reproduction). Reproduction times only included time in which participants' attention was engaged on reproduction, which we determined by observing changes in window focus, mouse positioning, and interface modality.

7.6 Results

Below, we summarize our findings of how Timelapse affects developers' reproduction behavior (RQ1) and how this interacts with debugging expertise (RQ2).

Timelapse did not reduce time spent reproducing behaviors. There was no significant difference in the percentage of time spent reproducing behaviors across conditions and tasks. Though Timelapse makes reproduction of behaviors simpler, it does not follow that this fact will reduce overall time spent on reproduction. We observed the opposite: because reproduction with Timelapse was so easy, participants seemed more willing to reproduce behaviors repeatedly. A possible confound is that behaviors in our tasks were fairly easy to reproduce, so Timelapse only made reproduction less tedious, not less challenging. We had hoped to test whether Timelapse is more useful for fixing more challenging bugs, but were forced to reduce task difficulty so that we could retain a within-subjects study design while minimizing participants' time commitment.

8–25% of time was spent reproducing behavior. Even when provided detailed and correct reproduction steps, developers in both conditions spent up to 25% (and typically 10–15%) of their time reproducing behaviors. Participants in all tasks and conditions reproduced behavior many times (median of 22 instances) over small periods. This suggests that developers frequently digress from investigative activities to reproduce program behavior. These measures are unlikely to be ecologically valid because most participants did not complete all tasks, and time spent on reproduction activities outside of the scope of our study tasks (i.e., during bug reporting, triage, and testing) is not included.

Expert developers incorporated replay capabilities. High-performing participants—those who successfully completed the most task steps—seemed to better integrate Timelapse's capabilities into their debugging workflows. Corroborating the results of previous studies [99, 115], we observed that successful developers debugged methodically, formed and tested debugging hypotheses using a bisection strategy, and revised their assessment of the root cause as their understanding of the defect grew. They quickly learned how to use Timelapse to facilitate these activities. They used Timelapse to accelerate familiar tasks, rather than redesigning their workflow around record/replay capabilities. In the Colorpicker task, participant 1 used Timelapse to step through each change to the widget's appearance to isolate a buggy RGB value. Participants in the control condition

appeared to spend much more time finding this buggy value, since they had to isolate a small change by interacting carefully with the widget. Participant 11 used Timelapse to compare program state before and after each call in the `mousemove` event handler, and then used Timelapse to move back and forth in time when bisecting the specific calls that caused the widget's RGB values to update incorrectly. Participants in the control condition appeared to achieve the same strategy more slowly by interleaving changes to breakpoints and manual reproduction.

Timelapse distracted less-successful developers. Those who only achieved partial or limited success had trouble integrating Timelapse into their workflow. We partially attribute this to differences in participants' prior debugging experiences and strategies. The less successful participants used ad-hoc, opportunistic debugging strategies; overlooked important source code or runtime state; and were led astray by unverified assumptions. Consequently, even when these developers used Timelapse, they did not use it to a productive end.

7.7 Discussion

TODO expand In our study, developers used Timelapse to automatically reproduce program behavior during debugging tasks, but this capability alone did not significantly affect task times, task success, or time spent reproducing behaviors. For developers who employed systematic debugging strategies, Timelapse was useful for quickly reproducing behaviors and navigating to important program states. Timelapse distracted developers who used ad-hoc or opportunistic strategies, or who were unfamiliar with standard debugging tools. Timelapse was used to accelerate the reproduction steps of existing strategies, but did not seem to affect strategy selection during our short study. As with any new tool, it appears that some degree of training and experience is necessary to fully exploit the tool's benefits. In our small study, the availability of Timelapse had no statistically significant effects on participants' speed or success in completing tasks. Figure 7.2 shows task success and task time per task and condition. In future work, we plan to study how

long-term use of Timelapse during daily development affects debugging strategies. We also plan to investigate how recordings can improve bug reporting practices and communication between bug reporters and bug fixers.

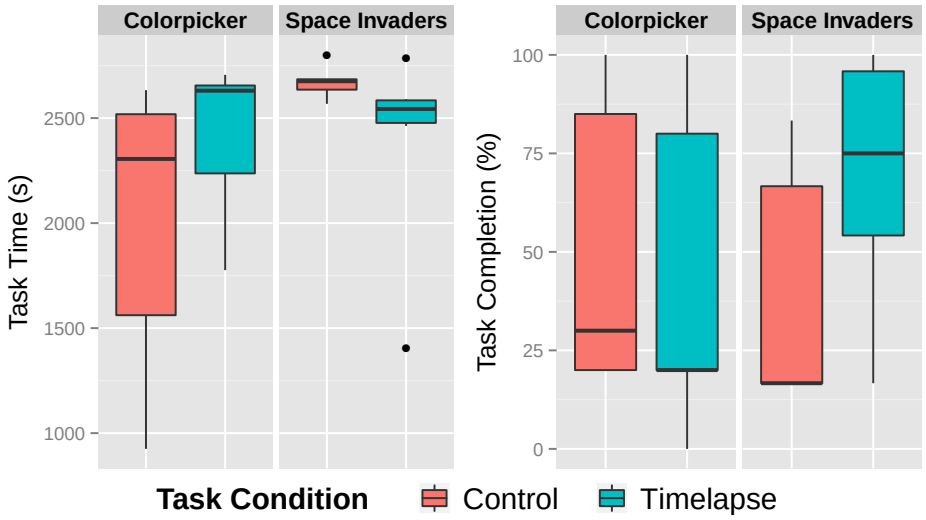

Figure 7.2: A summary of task time and success per condition and task. Box plots show outliers (points), median (thick bar), and 1st and 3rd quartiles (bottom and top of box, respectively). There was no statistically significant difference in performance between participants who used standard tools and those who had access to Timelapse.

Chapter 8
FUTURE WORK

Recordings produced by deterministic replay systems such as Dolos are small, but when executed, they produce vast amounts of runtime data that could be useful for many software engineering purposes. This chapter proposes new tasks and contexts which could significantly benefit from the capability to revisit past program states on demand. In particular, I explore a few applications of ubiquitous deterministic replay for software engineering: supporting collaborative bug reporting and diagnosis; synthesizing tests from recordings; supporting user-driven interactive dynamic analysis; exploring execution variations; and making it easier to empirically observe web program behavior in the large.

This chapter does not address short-term work items and long-term architectural changes that are necessary to generalize and integrate this dissertation's contributions into existing production systems. In the short term, adopting deterministic replay is largely a matter of addressing the shortcomings of research prototypes through software engineering. Potential improvements to the reliability, features, and architecture of this dissertation's prototypes are documented in the relevant chapters, and are not further discussed here.

8.1 Collaborative Debugging

Developers often cannot fix bugs in widely deployed software because they lack the means to reproduce the bug, or in many cases, even collect rudimentary diagnostic information that may help them fix the bug. Deterministic replay can make it easier for end-users to report bugs in web programs, and for web developers to fix reported bugs using a collaborative debugging workflow. Common to both of these visions is the need to

integrate and extend prior research into how recordings can anonymized and minimized. Clouse and Orso [30] investigated how to anonymize sensitive, nonessential user information in field failure reports. Jin and Orso [59] investigated how to remove inessential data from field failure reports (further discussed in Section 8.2.3).

Even when a failure can be reproduced reliably by a web developer, it is often difficult for a single developer to share their progress or seek help in diagnosing a failure. When possible, developers often prefer to synchronously interrupt a colleague to seek their help during debugging rather than post comments on an issue tracking system for later asynchronous feedback [11, 67]. I hypothesize that this tendency is partly a response to the high cost of context-switching to a debugging task. To resume a debugging task, a developer must recreate the failure, re-read and understand previous hypothesis and observations, and only then start making progress.

In addition to simplifying the process of recreating a failure, deterministic replay recordings can also enable a more documented and collaborative approach to debugging. If recordings were augmented with building blocks for collaboration such as annotations and versioning, then recordings could be the technical basis for a debugging "lab notebook". A debugging notebook would be shared amongst team members and contain the full history of a debugging session: important program states, executed commands, logged runtime values, developer notes, and associated code fixes. A notebook would contain the metadata necessary to revisit the relevant program instants mentioned in past entries. While the idea of a debugging notebook may seem far-fetched given the state of contemporary tools, this future is already a reality in the scientific computing community. IPython [100] and related tools allow users to create and share interactive notebooks that juxtapose code, results, and scientific analysis.

8.2 Creating Tests From Recordings

Using deterministic replay, it's possible to generate high-fidelity tests from a captured recording. In cases such as user interfaces, creating tests manually can be very time-

consuming. Many test environments rely on text descriptions of test cases. Using deterministic replay, a user can capture their interactions with a web program, interactively permute or change inputs to see their effects on execution, and extract input sequences that should be codified as an automated test. This is a more direct way of "authoring" a test because it avoids the need to abstractly express reproduction steps using a separate vocabulary of testing commands. An interactive test editor is included with STS [121][1], which allows users to author tests with specific interleavings that would otherwise be impossible to reliably trigger. This ability would be similarly useful for authoring tests that exercise interactive behaviors in web programs.

Tests derived from replayable recordings contain a subset of the inputs that constitute the recording; different types of tests use different subsets of a recording's inputs. A subset is necessary when deriving a test: if the full set of inputs were used, then the test would become fully deterministic, making behavior changes impossible. The choice of subset also determines what types of nondeterminism become possible. For example, a test generation tool might use a recording's user inputs (mouse events, scroll events, keyboard events, etc.) to synthesize a user interface test, but permit different program versions or event loop schedules. A tool that tests the throughput of a rendering engine could replay a recording as fast as possible, while permitting nondeterministic execution that does not cause the executing web program to diverge. A tool for minimizing a failure recording could discard inputs which are not causally related to a failure.

Below, I summarize some of the open research problems and promising approaches for generating different types of tests from deterministic replay recordings.

[1]STS is a fuzzer and simulator that finds bugs in software-defined networks (SDN). It uses deterministic replay to find failure-revealing event interleavings, and to explain causal sequence of events responsible for a crash.

8.2.1 User Interface Tests

Some of the most challenging tests to write are those that exercise interactive and dynamic behaviors. User interface tests are difficult to author because the functionality and interfaces that they exercise are highly visual and change frequently. UI tests also difficult to maintain: if tests are not written in a robust way, they will begin failing whenever anything substantial changes in the user interface. For example, a test that encodes user input events as a series of absolute screen coordinates is straightforward to create, but may fail if a different element is positioned at that coordinate during a later test run. To make tests less brittle to changes, prior work [**?**] has investigated strategies for uniquely and robustly identifying interface elements across program versions. These techniques could be used to synthesize robust tests from a captured recording.

TODO Add citations to testing-related literature above. See LaTeX comment for candidates.

8.2.2 Performance Regression Testing

Detecting performance regressions is critical when evaluating optimizations or other changes to a browser engine or JavaScript runtime. Browser vendors are especially interested in two performance metrics: latency and throughput. In the browser, *latency* (or time-to-first-paint) is the time required to download, parse, and render a web program to the screen. *Throughput* captures the rate over time (such as frames per second) at which a browser engine can respond to user actions and evaluate JavaScript, DOM code.

Measuring a browser's throughput is difficult because some of the most complex, important use cases are also the most challenging to exercise with automated tests. Automated tests cannot easily recreate what occurs when a user uses an interactive application such as Facebook, Gmail, or Google Docs. Server responses are nondeterministic and may change over time; even if network traffic is simulated using a replaying reverse-proxy [112, 122], nondeterministic JavaScript and DOM APIs can easily cause successive test executions to behave very differently. Due to the difficulty of testing these inter-

actions, browser vendors today rely on "non-interactive" DOM and JavaScript benchmarks[2]. TODO Make an appendix containing all the popular benchmarks. Dromaeo, v8, JSBench, Jetstream, Speedometer, PLT-2, Sunspider These short-running web programs are not representative of end-user browsing activity [112].

What if realistic performance tests could be synthesized from an execution captured by Dolos? In theory, this should be as straightforward as replaying a recording on different browser engine versions and collecting performance measurements. For example, browser vendors often want to track execution times for specific API calls, processing times for event loop tasks, overall memory usage, garbage collector activity, etc. In practice, browsers gain and lose functionality frequently, and these differences may cause a replayed execution to diverge. Web programs often use dynamic feature detection—testing at runtime whether an API is implemented—when deciding which code path or library implementation to use. Thus, exposing a different set of platform features and APIs to a replayed web program may cause different code to execute. Even if publicly visible APIs do not change between browser versions, changes to browser architecture or policies can impact internal browser engine nondeterminism and cause divergence. For example[3], event loop tasks could be scheduled differently in response to resource contention, resource cache policies could change, some work could be moved to its own thread or process, and so on.

A different approach to creating performance tests is to synthesize a best-effort deterministic test that does not rely on a deterministic browser engine. Richards et al. pioneered this approach with JSBench [112], a tool for generating JavaScript test scripts that approximate a captured browsing session. These tests are (necessarily) not completely deterministic, but their performance characteristics are stable enough across multiple browser versions to be used as a performance metric. Access to more precise recordings

[2]Browserbench: `https:///www.browserbench.org/`

[3]The changes mentioned here have occurred in upstream WebKit over the past few years during the course of my research. Each required nontrivial adjustments to the record/replay hooks used by Dolos.

may allow synthesized tests to be more consistent across browser engine versions.

8.2.3 Minimizing Recordings

Bug reports often contain irrelevant, misleading, or wrong information [148] that can hinder rather than help a developer in resolving an issue. The ability to submit a replayable recording produced by Dolos can reduce the amount of skill and effort required to report a bug (i.e., by writing reproduction steps, capturing output, or other steps). However, since recordings contain much more data than prose descriptions, recordings represent a potential *increase* in the amount of irrelevant information that a developer must investigate. Ideally, a recording submitted in lieu of reproduction steps would only contain the specific interactions and events necessary to reproduce a failure. But, even if a user creates a recording of themselves following reliable reproduction steps, they will unintentionally create user inputs—such as unhandled `mousemove` events—that become part of the recording but are not necessary to reproduce a failure.

Like program slicing and other techniques to aid comprehension, test case minimization techniques operate on the assumption that shorter recordings are strictly more useful because they reduce the amount of information that must be processed. This assumption seems reasonable, but it would be prudent[4] to obtain evidence to support this hypothesis before spending significant effort to develop minimization techniques whose usefulness is on it. In ongoing research outside the scope of this dissertation, my collaborators performed feasibility studies [51] to test whether this assumption is true in practice. In particular, we found that when traditional user interface test scripts [132] were minimized by removing irrelevant inputs, participants were able to diagnose and fix faults more quickly and successfully, supporting the assumption.

By thinking of a deterministic replay recording as an input file to a rendering engine,

[4]Many reasonable assumptions in programming languages research have been debunked with just a few months' of curiosity and skepticism. For example, contrary to widely-held assumptions, recent studies have found that web programs frequently use dynamic and reflective language features [111], and a list of suspicious statements does not help much in isolating a fault [99].

we can shorten recordings by applying well-known test case/input minimization [**?**] and delta debugging techniques [146]. Pruning irrelevant operations from a deterministic replay recording is inherently difficult because the strict invariants of fully deterministic execution are fragile. In the general case, dependencies between user inputs, network callbacks, and program data are extremely difficult to reason about using traditional program slicing techniques[5]. Scott et al. [121] have investigated a simpler alternative based on delta debugging and re-running to detect divergence. First, a specific input and its dependencies are removed from a recording based on domain knowledge or observations of coarse-grained causal dependencies. For example, Dolos could observe the relationship between an executed event loop task and the later event loop tasks that are spawned by it and use these dependencies when removing inputs. This approach is unsound because it does not consider fine-grained data dependencies in JavaScript code, but any divergences caused by disrupted data dependencies can be easily detected by re-running the reduced recording.

When reducing a recording, preserving a particular failure or behavior is often more important than maintaining strict determinism. For example, if a developer is only interested in the buggy behavior exhibited by a search bar widget, then inputs that affect other widgets on the page can be discarded. In this scenario, a recording reduction tool heavily depends on finding effective *test oracles* to guide delta debugging or other minimization algorithms. Finding effective, automated oracles for interactive behaviors and visual outputs is an active area of research [**?**].

8.3 On-demand, Retroactive Dynamic Analysis

The availability of deterministic replay could make many heretofore impractical program comprehension tools feasible for use with realistic programs and tasks. In his disserta-

[5] At the time of writing, I am not aware of any research prototypes that can usefully trace data and control dependencies through large-scale multiprocess software systems such as browsers or operating systems. If such a prototype existed and could identify irrelevant inputs, it would still be necessary to validate the removal via re-execution.

tion [32], Cornelissen identifies key intrinsic benefits and drawbacks of dynamic analysis for program comprehension: TODO make the following into a floated table, it destroys the flow of this paragraph

1. Benefit: *Enhanced precision* with respect to actual software behavior, such as executed control flow paths or call-site dispatch targets.

2. Benefit: The possibility for *goal-oriented strategies* which only analyse specific execution behavior: namely, features exercised through an execution scenario provided by the user.

3. Drawback: The analysis is inherently *incomplete*, only covering the small fraction of possible executions and runtime states covered by an execution scenario.

4. Drawback: It is difficult and important to find appropriate *execution scenarios* over which the analysis should be run.

5. Drawback: *Scalability* is a major concern due to the large amounts of data collected, stored, analyzed, and presented to the user.

6. Drawback: *Observer effects* can cause an instrumented program to no longer reproduce a behavior of interest, especially when the program uses multithreading or depends on precise execution timings.

The research literature contains thousands of papers describing dynamic analysis techniques which are infeasible for regular use because the drawbacks—especially *scalability*—outweigh the benefits. Most dynamic analysis techniques assume that an execution is transitory, so as the execution proceeds, an analysis must pessimistically collect all potentially relevant runtime state. However, excessive data collection is very frequently the limiting factor for scalability. For example, the Whyline for Java [65] can only capture

execution scenarios up to a few minutes in length due to excessive trace size. Scalability suffers because an analysis may add significant runtime slowdowns, consume too much main memory or disk, or induce observer effects. Applying principles from modern distributed systems and databases can increase the throughput for trace storage [106] and analysis [105], but this approach requires extensive engineering effort.

Deterministic replay could shift the benefit/drawback situation of dynamic analysis techniques to the point that many such techniques become feasible for realistic tasks and programs. This could be accomplished by several means: moving data collection and analysis in time in space; avoiding observer effects; improving completeness by combining executions; and designing entirely new multi-stage dynamic analysis techniques.

8.3.1 Improving Scalability and Reliability

The most straightforward improvement to scalability is the potential to decouple data collection and analysis from a specific execution [28, 29] or computing resource. This has clear usability benefits: a user can first capture an execution scenario using deterministic replay, and later perform the expensive instrumentation and analysis only if it is truly necessary. A user can run a dynamic analysis over a captured execution on a remote machine with more computing resources. (This is essentially the reverse of the solution proposed above for debugging field failures.) The initial recording phase incurs minimal overhead from deterministic record/replay, and subsequent re-executions—which are both deterministic and instrumented—are guaranteed to avoid observer effects.

The deterministic nature of Dolos and other record/replay infrastructures eliminates entire classes of observer effects. In particular, those caused by task interleavings or different timings are impossible because these aspects of execution must be carefully controlled by the infrastructure in order to achieve deterministic execution. Most observer effects caused by deterministic replay are related to the set of features that are unsupported or disabled during capture or playback. For example, Dolos disables certain in-memory and

OS-level resource caches during capture and replay. Most syscall-level [91, 119] or VM-level replay infrastructures [138] force all threads to execute on a single core during capture and replay. A dynamic analysis' instrumentation itself could be the source of some observer effects. For example, re-executing a recording with an instrumented browser may slow down an execution due to the extra memory usage and larger code size associated with instrumentation. If the instrumentation is transparent—that is, it doesn't diverge execution and is undetectable by client code—then observer effects caused by instrumentation are limited to time-related measurements.

8.3.2 Making Dynamic Analysis Interactive

A less understood direction for research is how deterministic replay enables new dynamic analysis techniques. Existing dynamic analysis techniques must pessimistically capture any potentially relevant data at runtime, on the assumption that executions are transitory. Some tools such as Whyline [65] initially collect runtime data in bulk, then perform post-hoc, on-demand data analysis to drive interactive user interfaces. What if data collection could be performed on-demand? How can a dynamic analysis collect data iteratively as it is needed, rather than all at once? How do user information needs map to configurations of instrumentation or an analysis?

To illustrate the potential of on-demand data collection, consider a scenario where a developer wants to find hot interprocedural execution *paths* and improve their performance. Existing tools provide complimentary capabilities that are useful in this scenario, but uses of these tools are mutually exclusive and require preemptive deployment. A tracing profiler [3] captures an exact calling context tree, but induces high runtime overhead that makes it unsuitable for profiling hot code. A sampling profiler [46] has very low runtime overhead, but only reports hot call stacks, obscuring paths through functions and fast or infrequently-executed code. Branch and path profilers [8] are designed to capture frequently executed sequences of basic blocks, but are even slower than tracing profilers.

Using iterative data collection, it would be possible to build a developer tool that mixes and matches the capabilities of these profilers to quickly identify hot interprocedural paths within a captured execution. Such a tool can accelerate a developer's workflow by only turning on profilers as necessary, and by configuring each profiler invocation to discard irrelevant information. First, the tool re-executes with a sampling profiler enabled to detect methods that dominate execution time. To identify related functions in the call graph that run quickly or are infrequently executed, the tool re-executes again using a tracing profiler. Finally, to detect the paths that intersect with user-selected functions or branches, the tool re-executes once more with a path profiler configured to discard paths that do not intersect the program point of interest. In effect, we can achieve the goals of adaptive [16] or "bursting" profilers [147]—only profile code that's interesting—by composing multiple uses of well-understood single-purpose profilers on successive playbacks.

Designing a replay-enabled dynamic analysis from scratch is hard because XXX, YYY, ZZZ. TODO expand What infrastructure and commands would support a replay-enabled dynamic analysis? TODO expand

TODO Move a short description of Overscan project idea to here?

8.4 Exploring Execution Variations

Developers often explore different program variations and execution variations. Talk about non-linear undo, exploring different design variations, other grant related stuff. TODO expand

Tool support is weak for both of these. Existing tools do blah, blah blah. TODO expand

What if a deterministic execution could have a nondeterministic ending? Then we could use this to support variations by doing XXX. TODO expand

How would controlled divergence work? rr does XXX. Viennot / Ladan / etc does XXX. STS [121] does XXX. Dolos could track coarse-grained causal dependencies and ignore internal nondeterminism, or something. TODO expand

8.5 A Database of Reusable Executions

Deterministic replay can enable large-scale corpus analysis of web program executions. Researchers and browser vendors frequently ask questions about how web programs behave at runtime: which browser and language features they use, whether optimizations pay off in practice, and whether a policy change would break existing web programs. In many cases, these questions can only be answered satisfactorily by analyzing many executions of real-world web programs. Recent research projects such as WebZeitgeist [69] have demonstrated the feasibility and utility of "mining" design elements from a corpus of thousands of static web pages. What if it were possible to mine *dynamic behaviors* from a corpus of thousands of captured executions?

Recent efforts to standardize the next version of JavaScript illustrate the importance of wide-scale impact analysis. In one instance[6], several implementors disagreed as to whether introducing a new method, `Array.prototype.contains`, would break web programs that make use of MooTools (a popular JavaScript utility library). At the time, MooTools was used by 6% of all public websites. Implementors were vexed: they can't afford to potentially break millions of websites, nor can they fix a library deployed on 3rd-party web servers, nor could they obtain real data as to whether this hypothetical corner case was an actual issue. In the end, implementors conservatively renamed the `contains` methods in `String` and `Array` to `includes`, breaking with the naming convention used by other new data types `Map` and `Set` and standard libraries in other languages.

With existing browser technology, the only way to answer many of these questions with runtime data is to manually interact with web programs using an instrumented browser. In prior work [111, 113] that analyzed whether and how often web programs used dynamic language features, my co-authors and I manually collected over 10,000 traces by browsing hundreds of websites using a specially-instrumented browser. This took over 2 man-weeks of effort to produce results for each paper. This manual approach

[6] **https://github.com/tc39/Array.prototype.includes**

is also very brittle: all previously gathered traces are rendered useless if instrumentation was incorrect or missing. It also harms reproducibility of research results: in this model there is no way for someone to verify that the traces correspond to reasonable user interactions. The same corpus cannot be reused across multiple research projects, resulting in related but incomparable results.

If a corpus of executions are captured as recordings using Dolos, then trace data could be regenerated as needed by replaying recordings using an instrumented browser. Traces can be produced via any appropriate mechanism, such as through an extension, injected page content, wrapping a browser view inside a tracking harness, or by instrumenting the browser engine itself. Traces produced by manual interaction or deterministic re-execution should not significantly differ, with some exceptions. For example, a trace may differ if: instrumentation causes execution to significantly diverge (**??**); the execution depends on unsupported platform features (**??**); or the trace captures aspects of execution that are intentionally left nondeterministic (**??**).

Automatically generating executions that are representative of real user interactions and browsing behavior is an active area of research [87]. However, even without automatic generation, using Dolos recordings as a canonical representation of execution factors out repetitive interactions, making collection a one-time cost. With some improvements to how recordings are shared, versioned, and anonymized (**??**), the task of capturing manual interactions could be distributed to any population of consenting browser users. Instead of performing analysis-specific data collection on an end-user's machine (as proposed by Liblit [82]), data is collected on-demand by re-executing an end-user's browsing session.

Chapter 9
CONCLUSION

This dissertation provides a glimpse into what is possible when deterministic replay capabilities are pervasive, transparent, and integrated into the runtime platform itself. In the course of performing the research described by this dissertation, I have demonstrate the huge of replay infrastructures for program comprehension so that their value is more widely appreciated by platform and tool developers.

This dissertation develops several enabling technologies for retroactively understanding dynamic behavior. First, we propose Dolos, a novel deterministic replay infrastructure that targets the highly dynamic, visual, and interactive domain of web applications. Dolos is the first such infrastructure to adapt precise, low-overhead virtual machine replay techniques to modern browser runtimes. Second, we propose Timelapse, a suite of affordances and algorithms for discovering and navigating to important program states within a captured execution. Timelapse introduces the concept of *probe points* and *time-indexed events* as affordances for navigating through a recording via its output. Finally, we propose Scry.

Appendix A
RESEARCH PROTOTYPES AND DEMOS

As a tool-heavy dissertation, much of this work's intermediate output and external visibility is in the form of code, prototypes, and interactive demos. This section lists the major prototypes and demos developed as part of this dissertation. For each, I briefly explain its purpose, research results derived, how it relates to other repositories or prototypes, and where the code may be obtained.

A.1 Prototypes

A.1.1 Pre-Timelapse (2010–2011)

Early experiments in capturing and replaying web content took place in branch of the DynJS [114] fork of WebKit in May 2011. Upstream changes were occasionally merged to the branch in large batches. This happened increasingly less frequently due to invasive instrumentation of the JavaScript interpreter and increasing complexity of the JavaScript runtime.

A.1.2 `timelapse-hg`[1] *(2011–2012)*

A new fork of WebKit, initially codenamed Timelapse[2], was created in Autumn 2011. This repository[1] contained a clean reimplementation of previous deterministic replay experiments. This prototype was the first to save nondeterministic inputs to a file, integrate

[1] `timelapse-hg` on Bitbucket: **`https://www.bitbucket.org/burg/timelapse/`**

[2] To distinguish the separate technical and design contributions, I later gave the Timelapse moniker to the user interface, and named the replay infrastructure Dolos. Replay functionality in mainline WebKit is referred to as *Web Replay* in design documents [21] and `WEB_REPLAY` in code.

with the Web Inspector interface, and shares many of the same architectural choices as more recent prototypes. This prototype was the basis for the first (rejected) conference submission describing Timelapse.

Unlike in later designs, this early prototype dispatched event loop inputs preëmptively and asynchronously. It counted the number of dispatched DOM events to know when to preëmptively inject the next event loop input. This was required to support the network replay machinery, which was an out-of-process reverse network proxy that captured and replayed browser network traffic. After months of dealing with unexplainable divergence and deadlock bugs, both of these approaches were abandoned in later prototypes for easier-to-debug mechanisms described in **??**.

As the first replay prototype with a user interface (Figure A.1, Figure A.2) and Web Inspector integration, a lot of time was spent trailblazing technical issues and learning how to implement basic interface functionality. With the pressure to publish our findings in order to get feedback on Timelapse's research contributions, there was little time to gain practical experience with the interface and use that to iterate on its design concepts until later. This interface featured an overview with a row of timelines per input category, and a sortable table containing details for each input.

A.1.3 `timelapse-git`[3] *(2012–2013)*

The second major prototype was reimplemented from scratch as a fork of WebKit, this time in a git repository[3] based on the official WebKit git mirror[4]. This prototype was used for the user studies in Chapter 7, so it received a lot of interface refinement and bug fixes. The visualization was refined to be more scalable and clearly show bursts of activity. The timeline (Figure 5.2 encodes input density over time using bubble radius and fill opacity, with higher intensity shown as larger and darker bubbles. The interface gained an overview timeline that showed relative composition of input types. In addition to these

[3] `timelapse-git` on GitHub: **`https://www.github.com/burg/timelapse/`**

refinements, many features and buttons were added to support scanning, bookmarks, and other integrations described in **??**. This prototype was the basis for the paper published at UIST [22].

A.1.4 Interface Redesign (2013)

The third major prototype was based on the same replay infrastructure but featured a redesigned interface (**??**) to match the redesigned Web Inspector. This new design abandoned an input-centric visualization in favor of a minimalist timeline overview and transparent integration with the Web Inspector's timeline interface. This prototype was the basis for probes and time-indexed outputs described in Chapter 4.

A.1.5 `replay-staging`[5] *(2013–2015)*

The last major prototype[6] focused on engineering and process improvements that made it easier to incorporate replay functionality into the upstream WebKit repository. Rather than maintaining a fork of WebKit, the `replay-staging` repository consisted of a patch series and a base WebKit commit. Instead of merging mainline changes into the fork, the patch series was rebased on top of new WebKit commits as functionality was contributed upstream. This prototype was the basis for the infrastructure extensions described in **??**.

While I interned at Apple Inc. in early 2014, over 100 patches were developed in this repository and incorporated into mainline WebKit. These changes included the core replay infrastructure [20] (described in **??**), a code generator for the input classes and file format used by Dolos, a testing harness for the Web Inspector, and general reliability and debugging improvements to the replay infrastructure. Separately from replay functionality, a production version of probe points (Section 4.2.1)

---

[4] WebKit `git` mirror: **`git://git.webkit.org/WebKit.git`**

[5] `replay-staging` on GitHub: **`https://www.github.com/burg/replay-staging/`**

A.1.6 Scry: `scry-staging`[7] *(2014–2015)*

The Scry prototype was developed as a separate patch series[7] on top of WebKit, similar to `replay-staging`. Scry was initially built on top of the `replay-staging` patch series to demonstrate that its capabilities could be integrated with Dolos. When this was achieved, Scry was moved to its own repository to make it easier to rebase on top of newer WebKit commits. This prototype was the basis for the tool described in Chapter 6. None of its functionality has been incorporated into WebKit at the time of writing.

A.2 Demos

While this dissertation's contributions are all embodied in interactive prototypes, with current tools [49] it is infeasible to create archival-quality packages of these software on Mac OS X, the primary development platform for WebKit. In lieu of software, this section describes short videos produced to demonstrate each major research project.

TODO Find a better way to archive the videos. Is there anything to say about them?

Timelapse[8]

Data Probes[9]

Scry[10]

---

[7] `scry-staging` on GitHub: **`https://www.github.com/burg/scry-staging/`**

[8] Timelapse Demo: **`https://www.dropbox.com/s/8uw1ufuox1bej3f/uist2013-cr.mp4?dl=0`**

[9] Probes Demo: **`https://www.dropbox.com/s/583b1sv3llchzux/chi2015-probes-submission-video.mp4?dl=0`**

[10] Scry Demo: **`https://www.dropbox.com/s/qh5yvi78n63gsn7/scry-uist-100m.mp4?dl=0`**

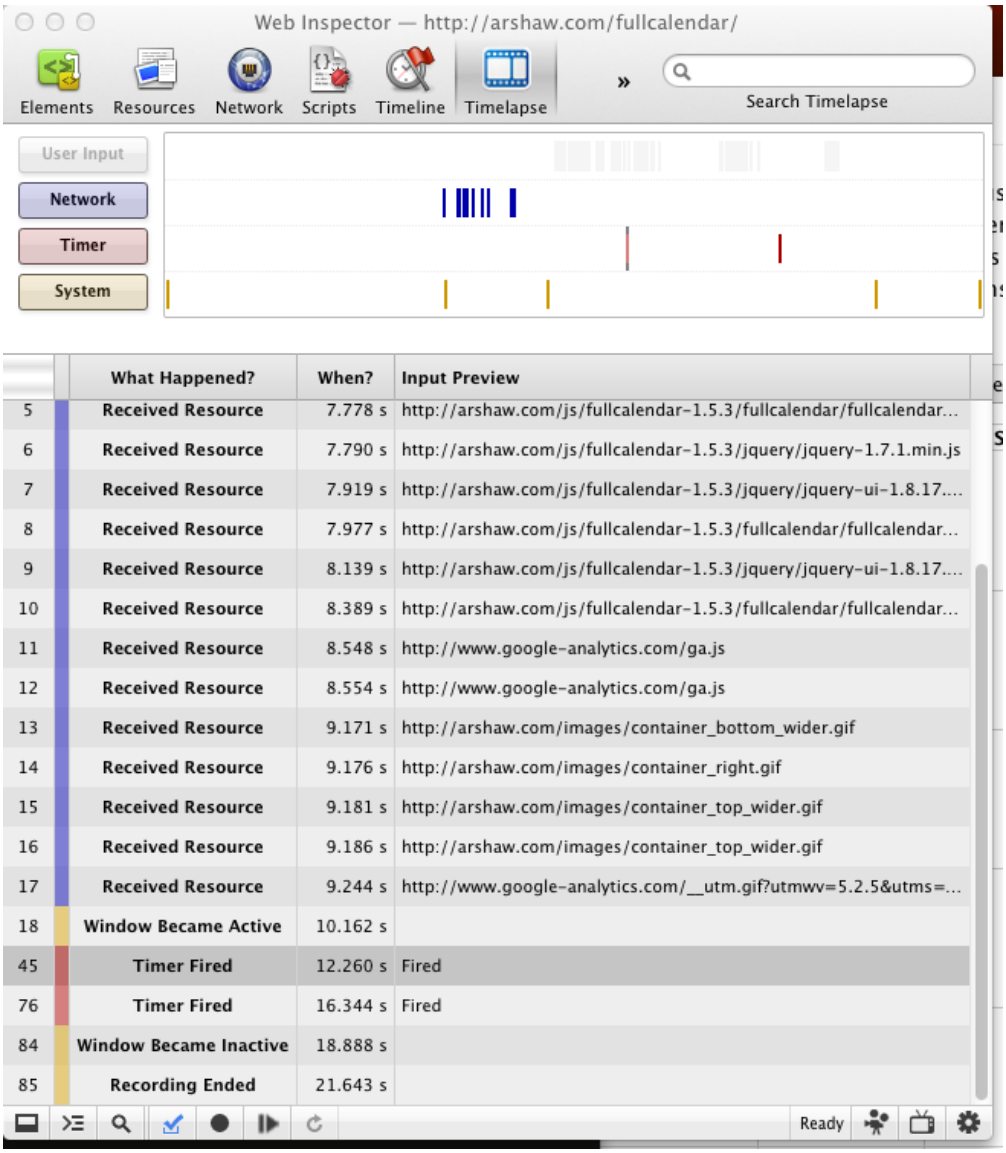

Figure A.1: An early prototype of Timelapse's input-centric timeline visualization. Users inputs are filtered out of the table.

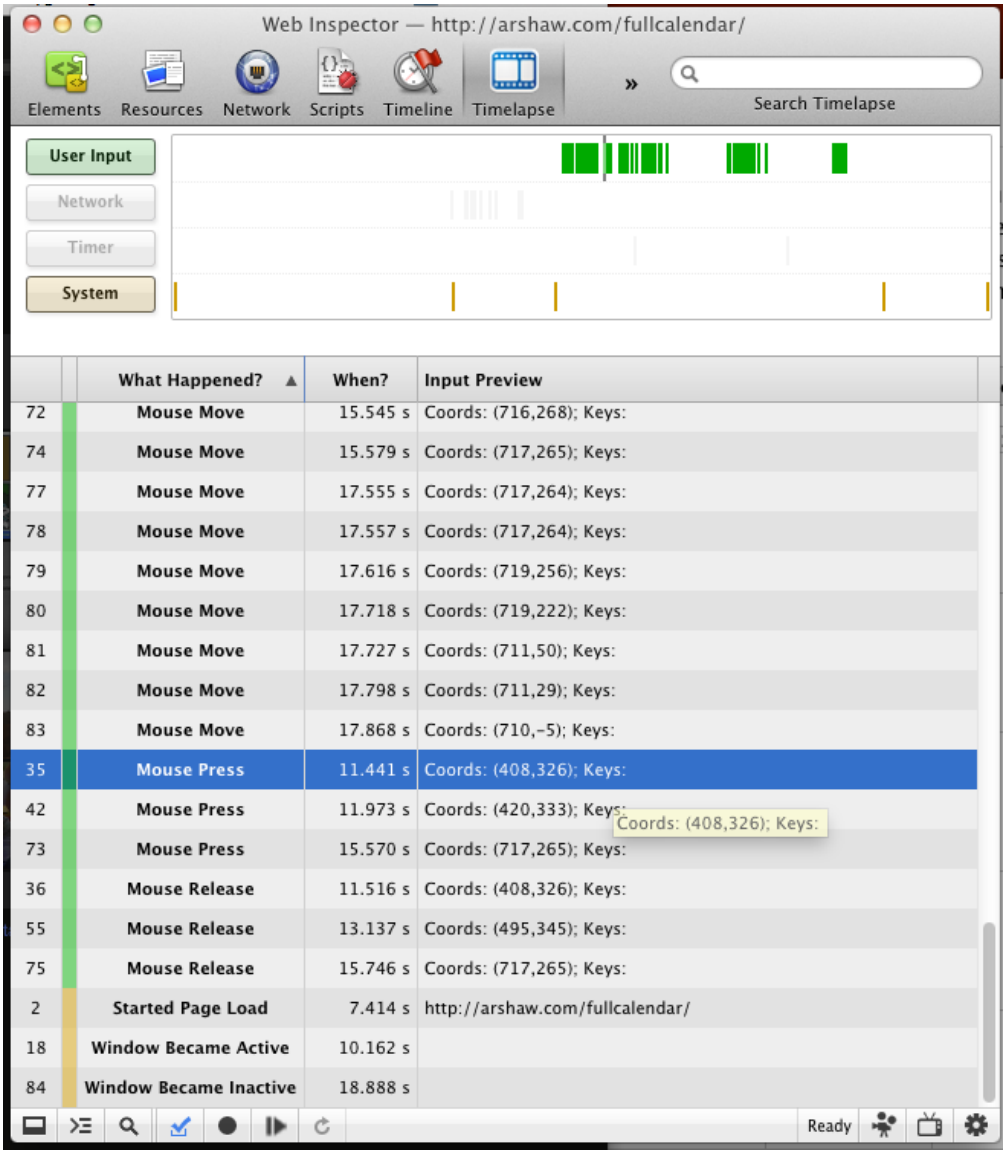

Figure A.2: An early prototype of Timelapse's input-centric timeline visualization. Network and timer inputs are filtered out, and the table is sorted by input type.

BIBLIOGRAPHY

[1] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding JavaScript event-based interactions. In *ICSE'14, Proceedings of the 36th International Conference on Software Engineering*, 2014. (Cited on pages 13, 19, and 85.)

[2] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009. (Cited on page 10.)

[3] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI 1997, Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, 1997. (Cited on page 107.)

[4] B. Anderson, L. Bergstrom, D. Herman, J. Matthews, K. McAllister, M. Goregaokar, J. Moffitt, and S. Sapin. Experience report: Developing the Servo web browser engine using Rust. *CoRR*, abs/1505.07383, 2015. URL **http://arxiv.org/abs/1505.07383**. (Cited on page 25.)

[5] S. Andrica and G. Candea. WaRR: A tool for high-fidelity web application record and replay. In *DSN'11: The 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2011. (Cited on pages 10 and 63.)

[6] D. Ansaloni, W. Binder, C. Bockisch, E. Bodden, K. Hatun, L. Marek, Z. Qi, A. Sarimbekov, A. Sewe, P. Tůma, and Y. Zheng. Challenges for refinement and composition of instrumentations: Position paper. In *Proceedings of the 11th International Conference on Software Composition*, 2012. (Cited on page 16.)

[7] K. Arya, T. Denniston, A.-M. Visan, and G. Cooperman. Semi-automated debugging via binary search through a process lifetime. In *PLOS '13: Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, Farmington, PA, USA, 2013. (Cited on page 12.)

[8] T. Ball and J. Larus. Efficient path profiling. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, 1996. (Cited on page 107.)

[9] BCEL Contributors. Apache Commons BCEL, 2014. **`http://commons.apache.org/proper/commons-bcel/`**. (Cited on pages 14 and 15.)

[10] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *USENIX 9th Symposium on OS Design and Implementation*, 2010. (Cited on pages 27 and 28.)

[11] D. Bertram, A. Voida, S. Greenberg, and R. Walker. Communication, collaboration, and bugs: The social nature of issue tracking in small, collocated teams. In *The 2010 ACM Conference on Computer Supported Cooperative Work*, 2010. (Cited on page 99.)

[12] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *The Second International Conference on Virtual Execution Environments*, 2006. (Cited on pages 11, 14, and 17.)

[13] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in WebKit's JavaScript bytecode. In *Proceedings of the 3rd Conference of Security and Trust*, 2014. (Cited on page 14.)

[14] P. Bille. A survey on tree edit distance and related problems. *Theoretical Comput. Sci.*, 337, June 2005. (Cited on page 76.)

[15] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. ORBS: language-independent program slicing. In *FSE 2014, Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*, 2014. (Cited on page 84.)

[16] M. D. Bond and K. S. McKinley. Practical path profiling for dynamic optimizers. In *PLDI 2005, Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, IL, USA, 2005. (Cited on page 108.)

[17] B. Boothe. Efficient algorithms for bidirectional debugging. In *PLDI 2000, Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, Vancouver, BC, Canada, 2000. (Cited on page 12.)

[18] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. Laviola Jr. Code Bubbles: rethinking the user interface paradigm of integrated development environments. In *ICSE'10, Proceedings of the 32nd International Conference on Software Engineering*, Cape Town, South Africa, 2010. (Cited on page 18.)

[19] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2009. (Cited on pages 20, 21, and 62.)

[20] B. Burg. Announcement: web replay support, 2014. **`https://lists.webkit.org/pipermail/webkit-dev/2014-January/026062.html`**. (Cited on page 114.)

[21] B. Burg. The mechanics of web replay, 2014. **`http://trac.webkit.org/wiki/WebReplayMechanics`**. (Cited on page 112.)

[22] B. Burg, R. J. Bailey, A. J. Ko, and M. D. Ernst. Interactive record and replay for web application debugging. In *Proceedings of the 26th ACM Symposium on User Interface Software and Technology*, 2013. (Cited on pages 22, 41, 52, 63, 74, 85, 89, and 114.)

[23] B. Burg, K. Madonna, A. J. Ko, and M. D. Ernst. Interactive navigation of captured executions via program output. Unpublished Draft, 2014. (Cited on page 41.)

[24] B. Burg, A. J. Ko, and M. D. Ernst. Explaining visual changes in web interfaces. In Review, 2015. (Cited on page 62.)

[25] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the 2004 USENIX Annual Technical Conference*, 2004. (Cited on pages 15 and 16.)

[26] T. Cardenas, M. Bastea-Forte, A. Ricciardi, B. Hartmann, and S. R. Klemmer. Testing physical computing prototypes through time-shifted and simulated input traces. In *Proceedings of the 21st ACM Symposium on User Interface Software and Technology*, 2008. (Cited on pages 9 and 10.)

[27] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko. Let's go to the whiteboard: How and why developers use drawings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2007. (Cited on page 20.)

[28] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Annual Technical Conference*, 2008. (Cited on pages 16, 48, 85, and 106.)

[29] J. Chow, D. Lucchetti, T. Garfinkel, G. Lefebvre, R. Gardner, J. Mason, S. Small, and P. M. Chen. Multi-stage replay with Crosscut. In *The 2010 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2010. (Cited on pages 17, 85, and 106.)

[30] J. Clouse and A. Orso. Camouflage: Automated anonymization of field data. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*, 2011. (Cited on pages 10 and 99.)

[31] F. Cornelis, A. Georges, M. Christiaens, M. Ronsse, T. Ghesquiere, and K. de Bosschere. A taxonomy of execution replay systems. In *In Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003. (Cited on page 9.)

[32] B. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. PhD thesis, Technische Universiteit Delft, 2009. (Cited on page 105.)

[33] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 36, September/October 2009. (Cited on page 13.)

[34] W. De Pauw and S. Heisig. Visual and algorithmic tooling for system trace analysis: a case study. *SIGOPS Operating Systems Review*, 44:97–102, January 2010. (Cited on page 19.)

[35] R. DeLine and K. Rowan. Code Canvas: Zooming towards better development environments. In *ICSE'10, Proceedings of the 32nd International Conference on Software Engineering*, 2010. (Cited on page 18.)

[36] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss. Debugger Canvas: industrial experience with the Code Bubbles paradigm. In *ICSE'12, Proceedings of the 34th International Conference on Software Engineering*, 2012. (Cited on page 18.)

[37] M. Denker, S. Ducasse, A. Lienhard, and P. Marschall. Sub-method reflection. *Journal of Object Technology*, 6(9):275–295, 2007. (Cited on page 15.)

[38] F. Détienne. *Software Design - Cognitive Aspects*. Springer-Verlang, 2001. (Cited on page 19.)

[39] C. Dionne, M. Feeley, and J. Desbiens. A taxonomy of distributed debuggers based on execution replay. In *In Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications*, 1996. (Cited on page 9.)

[40] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Operating Systems Review*, 36(SI):211–224, December 2002. (Cited on pages 9, 27, and 89.)

[41] C. Eckert and M. Stacey. Sources of inspiration: a language of design. *Design Studies*, 21(5):523–538, 2000. (Cited on page 62.)

[42] S. G. Eick, J. L. Steffen, and E. E. Sumner Jr. SeeSoft—a tool for visualizing line oriented software metrics. *IEEE Transactions on Software Engineering*, 18:957–968, November 1992. (Cited on page 19.)

[43] Facebook. React: a JavaScript library for building user interfaces, 2015. **`https://facebook.github.io/react/index.html`**. (Cited on page 82.)

[44] GDB Contributors. Inferiors and programs, 2014. **`http://sourceware.org/gdb/onlinedocs/gdb/Inferiors-and-Programs.html`**. (Cited on page 10.)

[45] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, 2005. (Cited on page 17.)

[46] S. L. Graham, P. B. Kessler, and M. K. McKusick. An execution profiler for modular programs. *Software: Practice and Experience*, 13:671–685, 1983. (Cited on page 107.)

[47] T. Grossman, J. Matejka, and G. Fitzmaurice. Chronicle: capture, exploration, and playback of document workflow histories. In *Proceedings of the 23rd ACM Symposium on User Interface Software and Technology*, 2010. (Cited on page 12.)

[48] P. J. Guo. Online Python Tutor: embeddable web-based program visualization for CS Education. In *SIGCSE '13: Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, 2013. (Cited on page 18.)

[49] P. J. Guo and D. Engler. CDE: Using system call interposition to automatically create portable software packages. In *Proceedings of the 2011 USENIX Annual Technical Conference*, 2011. (Cited on page 115.)

[50] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, W. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *USENIX 8th Symposium on OS Design and Implementation*, 2008. (Cited on pages 33 and 52.)

[51] M. Hammoudi, B. Burg, G. Bae, and G. Rothermel. On the use of delta debugging to reduce recordings and facilitate debugging of web applications. In *ESEC/FSE 2013: The 10th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2015. (Cited on pages 10 and 103.)

[52] C. M. Hancock. *Real-time programming and the big ideas of computational literacy*. PhD thesis, MIT Media Lab, 2003. (Cited on page 49.)

[53] J. Heer, J. D. Mackinlay, C. Stolte, and M. Agrawala. Graphical histories for visualization: supporting analysis, communication, and evaluation. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1189–1196, 2008. (Cited on page 12.)

[54] M. Hertzum and A. M. Pejtersen. The information-seeking practices of engineers: searching for documents as well as for people. *Information Processing & Management*, 36:761–778, 2000. (Cited on page 20.)

[55] M. Hilgart. Step-through debugging of GLSL shaders, 2006. (Cited on page 63.)

[56] J. Hollan, E. Hutchins, and D. Kirsh. Distributed cognition: towards a new foundation for human-computer interaction research. *ACM Transactions on Computer-Human Interaction*, 7:174–196, 2000. (Cited on page 20.)

[57] J. Huang, P. Liu, and C. Zhang. LEAP: lightweight deterministic multi-processor replay of concurrent Java programs. In *FSE 2010, Proceedings of the ACM SIGSOFT 18th Symposium on the Foundations of Software Engineering*, 2010. (Cited on page 27.)

[58] D. F. Jerding and J. T. Stasko. The information mural: a technique for displaying and navigating large information spaces. In *InfoVis '95: Proceedings of the First Information Visualization Symposium*, 1995. (Cited on page 19.)

[59] W. Jin and A. Orso. BugRedux: Reproducing field failures for in-house debugging. In *ICSE'12, Proceedings of the 34th International Conference on Software Engineering*, 2012. (Cited on pages 10 and 99.)

[60] T. Karrer, J.-P. Krämer, J. Diehl, B. Hartmann, and J. Borchers. Stacksplorer: call graph navigation helps increasing code maintenance efficiency. In *Proceedings of the 24th ACM Symposium on User Interface Software and Technology*, 2011. (Cited on page 56.)

[61] J. Kato, S. McDirmid, and X. Cao. DejaVu: integrated support for developing interactive camera-based programs. In *Proceedings of the 25th ACM Symposium on User Interface Software and Technology*, Cambridge, MA, USA, 2012. (Cited on pages 10, 12, and 49.)

[62] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005. (Cited on page 52.)

[63] A. J. Ko. *Asking and answering questions about the causes of software behavior*. PhD thesis, Human-Computer Interaction Institute, School of Computer Science, Carnegie Mellon University, 2008. (Cited on page 22.)

[64] A. J. Ko and B. A. Myers. Designing the WhyLine: a debugging interface for asking

questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2004. (Cited on page 85.)

[65] A. J. Ko and B. A. Myers. Extracting and answering why and why not questions about Java program output. *ACM Transactions on Software Engineering and Methodology*, 20(2):1–36, September 2010. (Cited on pages 11, 13, 14, 17, 18, 21, 42, 48, 50, 63, 71, 85, 105, and 107.)

[66] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, December 2006. (Cited on pages 21, 41, and 56.)

[67] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, 2007. (Cited on pages 20 and 99.)

[68] R. Kumar, J. O. Talton, S. Ahmad, and S. S. Klemmer. Bricolage: example-based retargeting for web design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2011. (Cited on pages 62, 76, and 83.)

[69] R. Kumar, A. Satyanarayan, C. Torres, M. Lim, S. Ahmad, S. R. Klemmer, and J. O. Talton. Webzeitgeist: Design mining the web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2013. URL **`http://vis.stanford.edu/papers/webzeitgeist`**. (Cited on page 109.)

[70] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocesor operating systems. In *ACM SIGMETRICS 2010: International Conference on Measurement and Modeling of Computer Systems*, New York, NY, USA, 2010. (Cited on page 10.)

[71] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *ICSE'10, Proceedings of the 32nd International Conference on Software Engineering*, Cape Town, South Africa, 2010. (Cited on page 21.)

[72] T. D. LaToza and B. A. Myers. Hard-to-answer questions about code. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, Revo, NV, USA, 2010. (Cited on pages 21 and 22.)

[73] T. D. LaToza and B. A. Myers. Designing useful tools for developers. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, Portland, OR, USA, 2011. (Cited on page 18.)

[74] T. D. LaToza and B. A. Myers. Visualizing call graphs. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, Pittsburgh, PA, USA, 2011. (Cited on page 21.)

[75] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *ICSE'06, Proceedings of the 28th International Conference on Software Engineering*, Shanghai, China, 2006. (Cited on page 20.)

[76] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2008. (Cited on page 21.)

[77] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering*, 39(2):197–215, February 2013. (Cited on pages 21, 63, and 69.)

[78] G. Lefebvre. *Composable and Reusable Whole-System Offline Dynamic Analysis*. PhD

thesis, University of British Columbia Department of Computer Science, Vancouver, BC, Canada, 2012. (Cited on page 17.)

[79] G. Lefebvre, B. Cully, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Tralfamadore: unifying source code and execution experience. In *EuroSys 2009*, 2009. (Cited on page 11.)

[80] B. S. Lerner, H. Venter, and D. Grossman. Supporting dynamic, third-party code customizations in JavaScript using aspects. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2010)*, 2010. (Cited on page 15.)

[81] B. Lewis. Debugging backwards in time. In *AADEBUG'2003, Fifth International Workshop on Automated and Algorithmic Debugging*, Ghent, Belgium, 2003. (Cited on pages 11, 12, 13, and 17.)

[82] B. R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of Wisconsin-Madison, 2004. (Cited on page 110.)

[83] T. Lieber, J. Brandt, and R. C. Miller. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Toronto, Canada, 2014. (Cited on page 17.)

[84] L. Marek, A. Villazón, D. Zheng, Yudi anad Ansaloni, W. Binder, and Z. Qi. DiSL: a domain-specific language for bytecode instrumentation. In *Proceedings of the 11th International Conference on Aspect-Oriented Software Development*, 2012. (Cited on pages 14 and 15.)

[85] L. Marek, S. Kell, Y. Zheng, B. Lubomír, W. Binder, P. Tůma, D. Ansaloni, A. Sarimbekov, and A. Sewe. ShadowVM: robust and comprehensive dynamic program analysis for the Java platform. In *Proceedings of the 12th International Conference on Generative Programming and Component Engineering*, 2013. (Cited on page 15.)

[86] S. McDirmid. Usable live programming. In *Proceedings of Onward! 2013*, 2013. (Cited on pages 12 and 49.)

[87] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling AJAX-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 6(1):1–30, 2012. (Cited on pages 13 and 110.)

[88] J. Mickens, J. Elson, and J. Howell. Mugshot: deterministic capture and replay for JavaScript applications. In *7th USENIX Symposium on Networked Systems Design and Implementation*, San Jose, CA, USA, 2010. (Cited on pages 9, 13, 32, 41, 52, 63, and 89.)

[89] P. Moret, W. Binder, and E. Tanter. Polymorphic bytecode instrumentation. In *Proceedings of the 10th International Conference on Aspect-Oriented Software Development*, 2011. (Cited on page 16.)

[90] Mozilla contributors. Mozilla Firefox web browser, 2014. **http://www.mozilla.org/en-US/firefox/desktop/**. (Cited on page 9.)

[91] Mozilla Contributors. rr: lightweight recording & deterministic debugging, 2014. **https://rr-project.org/**. (Cited on pages 9, 12, 27, and 107.)

[92] B. Myers, D. A. Weitzman, A. J. Ko, and D. H. Chau. Answering why and why not questions is user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2006. (Cited on pages 84 and 85.)

[93] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI 2007, Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, San Diego, CA, USA, 2007. (Cited on pages 11, 14, and 15.)

[94] M. W. Newman, M. S. Ackerman, J. Kim, A. Prakash, Z. Hong, J. Mandel, and T. Dong. Bringing the field into the lab: Supporting capturing and RePlay of con-

textual data for design. In *Proceedings of the 23rd ACM Symposium on User Interface Software and Technology*, 2010. (Cited on pages 9 and 10.)

[95] R. O'Callahan. Efficient collection and storage of indexed program traces, 2006. `http://www.ocallahan.org/Amber.pdf`. (Cited on pages 11, 17, and 63.)

[96] F. S. Ocariza Jr., K. Pattabiraman, and A. Mesbah. Vejovis: Suggesting fixes for JavaScript faults. In *ICSE'14, Proceedings of the 36th International Conference on Software Engineering*, 2014. (Cited on page 13.)

[97] J. Odvarko and Firebug contributors. Firebug: web development evolved, 2013. `http://www.getfirebug.com/`. (Cited on page 10.)

[98] S. Oney and B. Myers. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2009. (Cited on pages 9, 32, 63, and 72.)

[99] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA 2011, Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011. (Cited on pages 94 and 103.)

[100] F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007. ISSN 1521-9615. doi: 10.1109/MCSE.2007.53. URL `http://ipython.org`. (Cited on page 99.)

[101] M. Perscheid, B. Steinert, R. Hirschfeld, F. Geller, and M. Haupt. Immediacy through interactivity: online analysis of run-time behavior. In *2010 17th Working Conference on Reverse Engineering*, 2010. (Cited on page 16.)

[102] M. Petre, A. F. Blackwell, and T. R. G. Green. Cognitive questions in software visualization. In J. Stasko, J. Domingue, M. Brown, and B. Price, editors, *Software Visualization Programming as a Multi-Media Experience*, pages 453–480. MIT Press, January 1998. (Cited on page 20.)

[103] D. Piorkowski, S. D. Fleming, I. Kwan, M. Burnett, C. Scaffidi, R. Bellamy, and J. Jordhal. The whats and hows of programmers' foraging diets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2013. (Cited on page 21.)

[104] G. Pothier and E. Tanter. Extending omniscient debugging to support Aspect-oriented programming. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, Fortaleza, Ceará, Brazil, 2008. (Cited on pages 15 and 17.)

[105] G. Pothier and E. Tanter. Summarized trace indexing and querying for scalable back-in-time debugging. In *ECOOP 2011 — Object-Oriented Programming, 25th European Conference*, Lancaster, UK, 2011. (Cited on pages 11, 17, and 106.)

[106] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2007)*, Montreal, Canada, 2007. (Cited on pages 11, 13, 15, 17, 19, and 106.)

[107] Project Contributors. Debugging your program using Valgrind gdbserver and GDB, 2014. **`http://valgrind.org/docs/manual/manual-core-adv.html#manual-core-adv.gdbserver`**. (Cited on page 14.)

[108] P. Ratanaworabhan, B. Livshits, and B. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real Web applications. In *WebApps'10: USENIX Conference on Web Application Development*, 2010. (Cited on pages 11 and 17.)

[109] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *EuroSys 2009*, 2009. (Cited on page 25.)

[110] S. P. Reiss and M. Renieris. JOVE: Java as it happens. In *ACM Symposium on Software Visualization*, 2005. (Cited on page 16.)

[111] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI 2010, Proceedings of the ACM SIGPLAN 2010 Confer-*

ence on Programming Language Design and Implementation, 2010. (Cited on pages 11, 14, 17, 103, and 109.)

[112] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of JavaScript benchmarks. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2011)*, 2011. (Cited on pages 9, 13, 32, 101, and 102.)

[113] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: a large-scale study of the use of eval in JavaScript applications. In *ECOOP 2011 — Object-Oriented Programming, 25th European Conference*, 2011. (Cited on pages 14 and 109.)

[114] G. Richards, J. Vitek, F. Z. Nardelli, A. Domurad, F. Meawad, C. Hammer, B. Burg, and S. Lebresne. Dynamics of JavaScript, 2015. **http://plg.uwaterloo.ca/~dynjs/**. (Cited on page 112.)

[115] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering*, 30(12): 889–903, December 2004. (Cited on pages 41 and 94.)

[116] D. Röthisberger. Querying runtime information in the IDE. In *Proceeding of the 2008 Workshop on Query Technologies and Applications for Program Comprehension*, 2008. (Cited on page 19.)

[117] D. Röthliberger, M. Härry, A. Villazón, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. Augmenting static source views in IDEs with dynamic metrics. In *25th IEEE International Conference on Software Maintenance*, 2009. (Cited on page 15.)

[118] D. Röthlisberger. *Augmenting IDEs with Runtime Information for Software Maintenance*. PhD thesis, Universität Bern, 2010. (Cited on pages 17 and 67.)

[119] Y. Saito. Jockey: a user-space library for record-replay debugging. In *AADEBUG'2005, Sixth International Symposium on Automated and Algorithmic Debugging*, 2005. (Cited on pages 33, 52, 89, and 107.)

[120] R. Salkeld, B. Cully, G. Lefebvre, W. Xu, A. Warfield, and G. Kiczales. Retroactive aspects: Programming in the past. In *WODA 2011: Ninth International Workshop on Dynamic Analysis*, 2011. (Cited on page 16.)

[121] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. B. Acharya, K. Zarifis, and S. Shenker. Troubleshooting blackbox SDN control software with minimal causal sequences. In *SIGCOMM 2014*, Chicago, IL, USA, 2014. (Cited on pages 100, 104, and 108.)

[122] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *ESEC/FSE 2013: The 9th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, St. Petersburg, Russia, 2013. (Cited on pages 13, 14, 15, 17, 48, 84, 85, and 101.)

[123] J. Sillito, G. C. Murphy, and K. D. Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34:434–451, 2008. (Cited on pages 20, 21, 22, and 56.)

[124] M.-A. Storey. Theories, methods, and tools in program comprehension: Past, present, and future. In *Proceedings of the 13th IEEE International Workshop on Program Comprehension*, 2005. (Cited on page 20.)

[125] M.-A. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. In *Proceedings of the 5th Workshop on Program Comprehension*, 1997. (Cited on page 20.)

[126] M.-A. Storey, C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. Musen. Shrimp views: an interactive environment for information visualization and navigation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2002. (Cited on page 19.)

[127] J. Stylos and B. A. Myers. Mica: a web-search tool for finding API components and examples. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2006. (Cited on page 62.)

[128] M. Taeumel, B. Steinert, and R. Hirschfeld. The VIVIDE programming environment: connecting run-time information with programmers' system knowledge. In *Proceedings of Onward! 2012*, 2012. (Cited on page 18.)

[129] E. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: spatial and temporal selection of reification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, 2003. (Cited on page 16.)

[130] E. Tanter, P. Moret, W. Binder, and D. Ansaloni. Composition of dynamic analysis aspects. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering*, 2010. (Cited on page 16.)

[131] E. Tanter, I. Figueroa, and N. Tabareau. Execution levels for aspect-oriented programming: Design, semantics, implementations, and applications. *Science of Computer Programming*, 80:311–342, 2014. (Cited on page 16.)

[132] The Selenium Project. Selenium WebDriver documentation, 2012. **`http://seleniumhq.org/docs/03_webdriver.html`**. (Cited on pages 10 and 103.)

[133] R. Toledo, P. Leger, and E. Tanter. AspectScript: expressive aspects for the web. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, 2010. (Cited on pages 13, 14, and 15.)

[134] A. P. Tolmach and A. W. Appel. Debugging Standard ML without reverse engineering. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, 1990. (Cited on page 12.)

[135] J. Trümper, J. Bohnet, and J. Döllner. Understanding complex multithreaded software systems by using trace visualization. In *SoftVis '10: Proceendinsg of the 2010 ACM Symposium on Software Visualization*, 2010. (Cited on pages 18 and 19.)

[136] N. Viennot, S. Nair, and J. Nieh. Transparent mutable replay for multicore debugging and patch validation. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013. (Cited on page 10.)

[137] A.-M. Visan, K. Arya, G. Cooperman, and T. Denniston. URDB: a universal reversible debugger based on decomposing debugging histories. In *PLOS '11: Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, Cascais, Portugal, 2011. (Cited on page 12.)

[138] I. VMWare. Replay debugging on Linux, October 2009. **http://www.vmware.com/pdf/ws7_replay_linux_technote.pdf**. (Cited on pages 9, 12, 27, 52, and 107.)

[139] A. von Mayrhauser and A. M. Vans. From code understanding needs to reverse engineering tool capabilities. In *CASE'93: Proceedings of the Sixth International Workshop on Computer-Aided Software Engineering*, 1993. (Cited on pages 20 and 21.)

[140] A. Walenstein. *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework*. PhD thesis, Simon Fraser University, Burnaby, Canada, 2002. (Cited on page 20.)

[141] WebKit contributors. The WebKit open source project, 2012. **http://www.webkit.org/**. (Cited on pages 31 and 80.)

[142] WebKit contributors. About Safari Web Inspector, 2014. **https://developer.apple.com/library/safari/documentation/AppleApplications/Conceptual/Safari_Developer_Guide/Introduction/Introduction.html**. (Cited on page 10.)

[143] K. Yit Phang, J. S. Foster, and M. Hicks. Expositor: scriptable time-travel debugging with first-class traces. In *ICSE'13, Proceedings of the 35th International Conference on Software Engineering*, San Francisco, CA, USA, 2013. (Cited on page 17.)

[144] S. Yoo, D. Binkley, and R. Eastman. Seeing is slicing: Observation based slicing of picture description languages. In *Proceedings of the 14th International Working Conference on Soource Code Analysis and Manipulation*, 2014. (Cited on page 84.)

[145] A. Zaidman, N. Matthijssen, M.-A. Storey, and A. van Deursen. Understanding AJAX applications by connecting client and server-side execution traces. *Empirical Software Engineering*, 18(2):181–218, April 2013. (Cited on page 37.)

[146] A. Zeller and R. Hildbrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 38(2), February 2002. (Cited on page 104.)

[147] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive context profiling. In *PLDI 2006, Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, Ottawa, Canada, 2006. (Cited on page 108.)

[148] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C. Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643, September 2010. (Cited on page 103.)

[149] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schrter, and C. Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643, September 2010. (Cited on page 22.)