# Reverb: Speculative Debugging for Web Applications

Ravi Netravali
UCLA

James Mickens
Harvard University

## ABSTRACT

Bugs are common in web pages. Unfortunately, traditional debugging primitives like breakpoints are crude tools for understanding the asynchronous, wide-area data flows that bind client-side JavaScript code and server-side application logic. In this paper, we describe Reverb, a powerful new debugger that makes data flows explicit and queryable. Reverb provides three novel features. First, Reverb tracks *precise value provenance*, allowing a developer to quickly identify the reads and writes to JavaScript state that affected a particular variable's value. Second, Reverb enables *speculative bug fix analysis*. A developer can replay a program to a certain point, change code or data in the program, and then resume the replay; Reverb uses the remaining log of nondeterministic events to influence the post-edit replay, allowing the developer to investigate whether the hypothesized bug fix would have helped the original execution run. Third, Reverb supports *wide-area debugging* for applications whose server-side components use event-driven architectures. By tracking the data flows between clients and servers, Reverb enables speculative replaying of the *distributed* application.

## KEYWORDS

record-and-replay debugging, systems debugging

**ACM Reference Format:**
Ravi Netravali and James Mickens. 2019. Reverb: Speculative Debugging for Web Applications. In *SoCC '19: ACM Symposium of Cloud Computing conference, Nov 20–23, 2019, Santa Cruz, CA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3357223.3362733

## 1 INTRODUCTION

Debugging the client-side of a web application is hard. The DOM interface [40], which specifies how JavaScript code interacts with the rest of the browser, is sprawling and constantly accumulating new features [27, 35]. Furthermore, the DOM interface is pervasively asynchronous and event-driven, making it challenging for developers to track causality across event handlers [26, 36, 49, 72]. As a result, JavaScript bugs are endemic, even on popular sites that are maintained by professional developers [57, 59].

Commodity browsers include JavaScript debuggers that support breakpoints and watchpoints. However, fixing bugs is still hard. Breakpoints and watchpoints let developers inspect program state at a moment in time; however, in an event-driven program with extensive network and GUI interactions, bug diagnosis often requires complex temporal reasoning to reconstruct a buggy value's *provenance* across multiple asynchronous code paths. This provenance data is not exposed by more advanced tools for replay debugging or program slicing (§2).

In this paper, we introduce Reverb, a new debugger for web applications. Reverb has three features which enable a fundamentally more powerful debugging experience. First, Reverb tracks *precise value provenance*, i.e., the exact set of reads and writes (and the associated source code lines) that produce each program value. Like a traditional replay debugger [13, 36, 62], Reverb records all of the nondeterministic events from a program's execution, allowing Reverb to replay a buggy execution with perfect fidelity. Unlike a traditional replay debugger, Reverb also records the *deterministic* values that are manipulated by reads and writes of page state. Using this extra information at replay time, Reverb enables developers to query fine-grained data flow logs and quickly answer questions like "Did variable `x` influence variable `y`?" or "Was variable `z`'s value affected by a control flow that traversed function `f()`?" Reverb's logging of both nondeterministic and deterministic events is fast enough to run in production: for the median web page in our 300 page test corpus, Reverb increases page load time by only 5.5%, while producing logs that are only 45.4 KB in size.

Reverb's second unique feature is support for *speculative bug fix analysis.* At replay time, Reverb allows a developer to pause the application being debugged, edit the code or data of the application, and then resume the replay. Post-edit, Reverb replays the remaining nondeterministic events in the log, using carefully-defined semantics (§3.3) to determine how those events should be replayed in the context of the edited program execution. Once the altered execution has finished replaying, Reverb identifies the control flows and data flows which differ in the edited and original executions. These analyses help developers to determine whether a hypothesized bug fix would have helped the original program execution. Speculative edit-and-replay is unsound, in the sense that a post-edit program can misbehave in arbitrary ways, e.g., by attempting to read an undefined variable. However, even without Reverb, *the process of testing bug fixes is unsound.* A developer typically lacks a priori knowledge about whether a hypothesized fix will work. The developer implements the hypothesized fix, and then runs tests and tries to determine whether the fix actually worked; even if all of the tests pass, there is no guarantee that the fix is completely correct, since the tests may miss corner cases. However, Reverb provides the developer with an important new weapon: the ability to compare the data flows and the control flows in the original execution and the ostensibly bug-free execution. As we demonstrate through case studies (§5.3), the ability to diff program executions is a powerful debugging tool.

Reverb's third novel feature is to support *wide-area debugging* for applications whose server-side components use single-threaded, event-driven architectures like Node [56], Redis [60], or

*SoCC '19, November 20-23, Santa Cruz, CA*

ACM ISBN 978-1-4503-6973-2/19/11...$15.00
https://doi.org/10.1145/3357223.3362733

NGINX [53]. For these components, the event loop interface provides a narrow, semantically-well-defined abstraction layer at which to log and replay the components. Thus, Reverb can use vector clocks and a small assortment of additional tricks (§3.4) to track wide-area causality. Reverb provides two levels of support for server-side components:

- Node components execute JavaScript code. Thus, Reverb can apply its client-side framework to the server-side, and track variable-level data flows and control flows between multiple browsers and multiple server-side Node instances.
- Reverb treats an event-driven (but non-JavaScript) component like Redis as a black box. Reverb logs and replays the component at the level of the component's externally-visible event interface, tracking data flows emanating from, and terminating at, server-side events.

Reverb supports speculative bug fix analysis for data stores and JavaScript state on either side of the wide area. For example, a developer can edit the value that server-side code receives from a Redis database, and then explore how the edited value impacts the remainder of the replaying application's execution.

In summary, our contribution is the first distributed replay debugger that provides fine-grained data flow tracking and speculative bug fix analysis. Supporting this entire set of debugging capabilities was previously intractable, because prior debuggers operated at the wrong semantic level; Reverb's insight is that web services using managed runtimes and event-driven cross-server RPCs *should be analyzed at these levels of abstraction*, instead of at the level of system calls or hardware-level instruction traces. However, to fully leverage this new insight, we must provide new debugging infrastructure that prior work lacks. In particular, we introduce a new logic for reasoning about post-edit replays; this logic describes how editing an application component mid-replay should affect the post-edit replay of that component (§3.3) and remote ones which may see altered output from the mutated component (§3.4). We also introduce new diagnostic techniques for helping developers understand how edited replays diverge from a program's original executions. These techniques, which explain divergences using diffs of data flow graphs and control flow graphs (§3.2), allow Reverb to diagnose complex bugs in real web applications (§5.3). A user study confirms that Reverb is more helpful than traditional in-browser debuggers (§5.6).

# 2 BACKGROUND

Having used the debuggers in commodity browsers [18, 21, 39], and having built several state-of-the-art debuggers ourselves [34, 36], we often found ourselves wanting fine-grained data flow tracking and speculative bug fix analysis. In this section, we explain why prior debugging techniques are insufficient to realize the vision of Figure 1.

## 2.1 Traditional debuggers

Standard debuggers focus on the abstraction of breakpoints [18, 21, 39]. Debuggers like Visual Studio [37] and Eclipse [14] also allow developers to edit some types of program values at a breakpoint, and then resume the program's live execution. Breakpoints are undoubtedly useful, but they force a human developer to guess which source code locations are buggy.

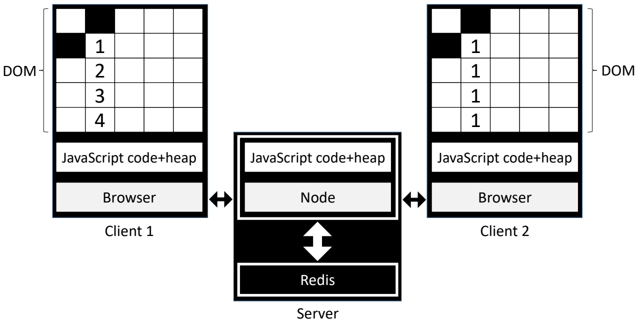


**Figure 1: EtherCalc [63] is a web-based, collaborative spreadsheet. Multiple users can simultaneously issue edits to the same spreadsheet, with a Node server broadcasting edits to all users, and storing the spreadsheet data in Redis. Bug #314 in EtherCalc's issue tracker involves a GUI-based edit from client 1 that is not reflected to client 2's DOM. Ideally, a debugging framework could efficiently answer two questions. First, how is the relevant DOM state and JavaScript heap state from client 1 being transmitted through the server-side components to the DOM and JavaScript heap of client 2? Second, given recorded state from a buggy execution run, as well as a hypothesized bug fix that modifies code and/or data on clients or servers, would the hypothesized fix remove the problematic behavior in the recorded execution?**

Some debuggers support watchpoints, which pause an application when a specific memory location is read or written. Watchpoints eliminate the need for a developer to guess when and where a particular buggy assignment will occur. However, watchpoints do not capture temporal data flows throughout a program. So, developers still have to manually reconstruct reverse temporal flows to determine how the value in a buggy write was generated. Our case studies (§5.3) demonstrate that automated construction of value provenance eliminates human-driven guess work about how program state is created.

## 2.2 Deterministic replay

Traditional debuggers pause and inspect the state of live programs. In contrast, replay debuggers [4, 13, 15, 19, 20, 29, 31, 36, 62] first *log* the nondeterministic events in a live execution run, and then *replay* the program in a controlled environment, using the log to carefully recreate the original order and content of nondeterministic events. Replaying the nondeterministic events is sufficient to induce the remaining, deterministic program behavior, so there is no need to log the values that are manipulated by deterministic reads and writes.

The ability to reliably recreate a buggy execution makes it easier to test fault hypotheses. Replay debugging is particularly useful for studying heisenbugs that rarely occur and involve specific event orderings. Some replay debuggers support backwards-stepping [15, 19, 31, 62], such that a developer can set a breakpoint or a watchpoint, and then move execution forwards or backwards in time. However, even backwards-stepping debuggers force human developers to manually track value provenance. Thus, root cause analysis is still difficult.

### 2.3 Program slicing

A program slice is a subset of program statements that may have influenced the values that are accessed by a specific line of source code [66, 69]. The tuple `<sourceCodeLine,variablesOfInterest>` is called the slicing criterion. Given a slicing criterion, a static slice is derived purely from analysis of source code [10, 12, 16, 24, 55]; in contrast, a dynamic slice assumes a set of concrete values (e.g., at the slicing criterion) to narrow the set of potentially relevant program statements [1, 2, 23, 30, 66].

Slicing algorithms lack a complete, concrete log of the reads and writes made during a real execution; thus, slicing algorithms are often imprecise, particularly for complex programs. Imprecision hurts the use of slices for bug diagnosis, since developers must consider source code lines that may not be causally related to the bug. Imprecision compounds itself if slices are used to reconstruct wide-area execution behavior. In contrast, Reverb provides *guaranteed-precise, provenance-annotated execution traces* (§3.2). Similar to an instruction trace, a Reverb trace provides a temporal log of the source code statements that a program executed; however, the traces also describe the *values* that the executed statements manipulated, and the *provenance* of those values. For additional discussion of program slicing, the interested reader can peruse Section A.4 in the technical report [5].

### 2.4 Data Provenance

Provenance-aware file systems [48, 61] allow users to determine which input files were read by a process during the production of output files. Reverb deals with the provenance of application state at the granularity of *individual program variables* that reside on clients and servers. Thus, Reverb tracks how storage data spreads throughout an application, but does so at the level of fine-grained, variable-level flows.

Provenance-aware network platforms let operators discover the route that a packet took [74], or the reason why network switches have certain NDlog rules [70, 71, 73]. Reverb is agnostic about network-level configuration state, but is compatible with systems that track it.

### 2.5 Speculative edit-and-continue

Dora [67] is a single-machine replay debugger that records the OS-level interactions that belong to a group of processes. Dora allows for limited types of edits to occur during replay. If an edit causes a replay to diverge, Dora explores multiple execution paths that are rooted at this initial divergence. Dora executes each post-divergence path on a live machine, recording the subsequent (and nondeterministic) OS-level interactions. Like Reverb, Dora defines policies for handling new calls to timekeeping functions or socket interfaces. After Dora has explored several potential futures of the divergent replay, Dora identifies the most plausible divergent execution using a metric akin to string edit distance, comparing the system calls of each explored path to those of the original execution.

Dora's speculative power is highly restricted by two factors. First, Dora's vantage point is at the OS layer. In contrast, Reverb's vantage point is within the managed runtime of a JavaScript engine, or at the event loop interface of a single-threaded program like Redis. This difference is fundamental, and represents a key insight of Reverb: *by introspecting on program execution at a higher level of abstraction, Reverb can handle a wider variety of speculative edits*, because the side effects of an edit can be reasoned about with respect to a constrained set of events, instead of the much wider and messier POSIX interface. For example, Dora cannot handle edits which modify thread scheduling, e.g., to cause fewer threads to run, because Dora cannot enumerate and model the ensuing avalanche of side effects upon low-level POSIX state like pthread locks and shared memory pages. In contrast, Reverb can handle a schedule-altering edit that changes the number of client-side frames (the JavaScript equivalent of processes). Reverb can tractably reason about such changes because frames cannot share raw memory, are internally single-threaded, and only communicate via pass-by-value `postMessage` events. Thus, the only way that a newly created frame can impact another frame is via the generation of new `postMessage` events.

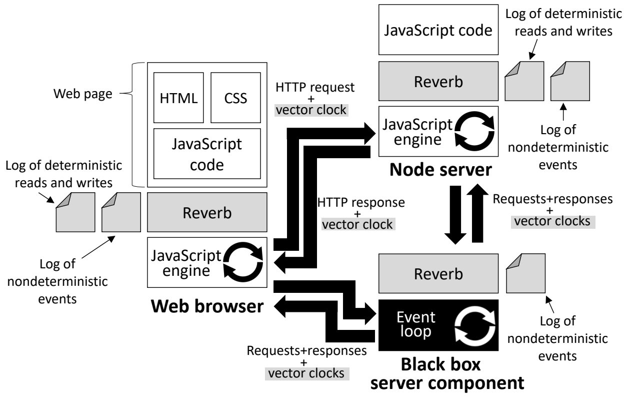

JavaScript code
Log of deterministic reads and writes
Reverb
Web page
HTML
CSS
JavaScript code
HTTP request + vector clock
JavaScript engine
Log of nondeterministic events
Node server
Log of deterministic reads and writes
Reverb
HTTP response + vector clock
Requests+responses + vector clocks
JavaScript engine
Reverb
Log of nondeterministic events
Web browser
Event loop
Log of nondeterministic events
Requests+responses + vector clocks
Black box server component

**Figure 2: An overview of Reverb's architecture. Grey components are added by Reverb.**

Dora's second restriction is that it does not track individual reads and writes to raw memory, because doing so would be too expensive [50]. Thus, Dora cannot provide variable-level value provenance; another consequence is that Dora may incorrectly replay post-edit memory accesses if the edit changes which memory page contains an object. In contrast, Reverb introspects at the JavaScript level, allowing Reverb to efficiently track all reads and writes to *application-visible* state. This difference is fundamental. Logging all reads and writes enables wide-area causality tracking, and is critical for explaining divergences between a logged program run and a speculatively-edited replay (§3.3).

## 3 DESIGN

Figure 2 provides an overview of Reverb's design. A web application has multiple clients and servers. Clients are assumed to be standard web browsers which execute JavaScript. Both server-side and client-side components are assumed to be single-threaded and event-driven. Each component records its nondeterministic events; if a component uses a JavaScript engine, then the component also records its deterministic reads and writes to JavaScript state and the DOM (§3.2). Distributed causality between hosts, e.g., via HTTP requests, is tracked using vector clocks (§3.4). At debug time, Reverb uses the global event log to replay each client or server in isolation, or together as a single logical application (§3.3 and §3.4).

### 3.1 Overview of the JavaScript Execution Model

**Execution environment:** JavaScript exposes a single threaded, event-driven programming interface. A JavaScript file defines initialization code that runs once, at the time that the file is parsed by the JavaScript engine. The initialization code registers event handlers that the JavaScript engine will fire in response to GUI interactions, timer expirations, network activity, and so on. Once a browser has evaluated all of the JavaScript files in a page's HTML, the subsequent execution of the page is driven solely by the reception of asynchronous events.

An event handler often calls other functions. Thus, firing a handler can initiate a call chain that is rooted by the handler. A program can register multiple handlers for a single event type. Thus, the call chain for an event is the union of the call chains for the associated event handlers. In the rest of this paper, the unadorned term "call chain" refers to the aggregate call chain for a particular event.

**Sources of Nondeterminism:** In a JavaScript program, the primary source of nondeterminism is the order in which events arrive (and the content of those events) [36]. JavaScript code may also invoke a small number of nondeterministic functions. For example, `Math.random()` returns a random number. `Date()` returns the current time with millisecond granularity.

By default, a JavaScript program consists of a single event loop. However, a web page can incorporate multiple frames [42] or web workers [25]; each one represents a new event loop that runs in parallel with the others [38]. Concurrent execution contexts can only interact with each other via the asynchronous, pass-by-value `postMessage()` interface [44]. The browser delivers those messages by firing an event in the recipient's execution context. Thus, from the perspective of the recipient, handling message nondeterminism is no different than handling other event-driven nondeterminism like GUI interactions.

**Externalizing Output:** A JavaScript program can externalize three types of output:

- The DOM interface [40] lets a program update the visual content that users see. The DOM interface defines methods for dynamically manipulating a page's HTML structure, e.g., by adding new HTML tags, or by changing the CSS styles of preexisting tags.
- A JavaScript program can also write to local storage. Cookies [41] can store up to 4 KB of data, whereas IndexedDB [43] and the `localStorage` interface [45] can hold MBs of information.
- To send network data, a program uses the `XMLHttpRequest` [47] and `WebSocket` [46] interfaces. `XMLHttpRequest` is an older interface which only supports request/response interactions. `WebSocket` supports full-duplex streams.

In this paper, we ignore multimedia objects like `<video>` streams, since we focus on the debugging of pure HTML, CSS, and JavaScript state.

### 3.2 Analyzing Value Provenance

To track data flows, Reverb first logs nondeterministic and deterministic events. After reconstructing data flows, Reverb uses them to support flow queries, and express state divergences caused by speculative edit-and-continue.

**Logging Nondeterminism:** Prior work has explored various ways to deterministically replay client-side JavaScript code [8, 36]. Our Reverb prototype rewrites JavaScript source code to interpose on nondeterministic sources (§4), but Reverb's design makes no deep assumptions about how nondeterminism is logged or replayed.

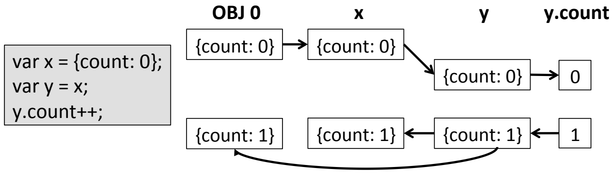
A Reverb log has an entry for each nondeterministic event; each log entry contains event-specific data that is sufficient for recreating that event. For example, the log entry for a timer firing contains a reference to the timer callback. The log entry for a call to `Math.random()` contains the return value of the function. The log entry for a mouse click stores which mouse button was clicked, the x and y coordinates for the click, and so on.

At the beginning of logging, Reverb takes a snapshot of the client's local storage (e.g., cookies). Reverb also registers its own handlers for GUI events like mouse clicks. So, if the logged application only installs handlers for (say) `keypress`, but not `keydown` or `keyup`, Reverb will still log when the latter two kinds of events occur. This information is useful for handling speculative edits which add new GUI handlers (§3.3).

**Logging Deterministic Reads and Writes:** In JavaScript, each object is essentially a mutable dictionary, with string keys (i.e., property names) mapping to property values. The global namespace is reified via the special `window` object, such that references to a global variable `x` are implicitly translated to `window.x`. Abstractly speaking, Reverb logs reads and writes to the JavaScript heap by shimming the getters and setters for each object dictionary (including the one that belongs to `window`). Our Reverb prototype uses JavaScript rewriting to inject this shim code (§4).

**Reconstructing Data Flows:** Using the log of deterministic reads and writes, Reverb can reconstruct the provenance of all JavaScript variable values at any moment in a program's execution. Given a slicing criterion which mentions variable `x` at time `t`, Reverb finds the prior write for which `x` was the left-hand side. For the variables on the right-hand side, Reverb finds the prior write which assigned to *those* variables. Reverb continues this recursive process until reaching the beginning of the program; the traced path represents the provenance for the slicing criterion. Note that the path may be a tree, not a line, because a single assignment may involve multiple right-hand sides (e.g., `x=y+z`). The path may also cross the event handlers that belong to multiple high-level events like key presses or the arrival of network data.

Reverb's log associates each deterministic read or write with a source code line. Thus, Reverb can also generate source-code-level execution traces which provide a serial history of each source code line that a program ran. The core visualization tool that Reverb provides to developers is an execution trace that is overlaid with provenance information: each variable mentioned in each source code line is associated with the prior source code line which generated the variable's value. Figure 7 depicts an example of such a trace, and the extended technical report [5] (§A.3) describes some of the pruning techniques which improve the comprehensibility of traces. For now, we merely explain a developer-guided pruning approach that is simple, and important in practice: targeted dynamic tracing. A targeted dynamic trace lets a developer drill down on the executed source code lines (and associated data flows) that affected a specific variable. As the developer explores the initial trace, the developer can



```
var x = {count: 0};
var y = x;
y.count++;
```

**Figure 3: To capture aliasing relationships, Reverb distinguishes between an underlying object and its multiple names. Writes to an aliased object create horizontal arrows in data flow diagrams, since time flows downward and the aliases are updated simultaneously.**

add or remove target variables, expanding or shrinking the targeted trace. Our case studies (§5.3) show that targeted dynamic traces are fast to generate, and provide helpful diagnostic information.

Reverb uses Scout-style dependency analysis [51] to track data flows between the JavaScript heap and the DOM. For example, the DOM tree is a data structure which mirrors a page's dynamic HTML tree; each HTML tag has an associated DOM node that is exposed to JavaScript code. Reverb understands the semantics of DOM methods like `Node.appendChild(newChild)`. Thus, Reverb can track how JavaScript values flow to DOM nodes, and how DOM values are assigned to JavaScript variables.

Reverb's logs capture a variety of additional behavior. For example, Reverb explicitly tracks aliasing relationships, as shown in Figure 3. Also, for each executed branch, Reverb records the associated source code line, and the values consumed by the branch test. This information allows Reverb to apply classical algorithms for building dynamic control flow dependencies [30]. Reverb easily handles the special case of execution flows that span `try/catch` blocks, since Reverb records both the exception-throwing line, and the catching line.

### 3.3 Speculative Edit-and-Continue Debugging

Speculative edit-and-continue debugging has five phases:

- logging the events in a baseline execution run;
- replaying the execution up to a specified point;
- changing the program's state in some way;
- resuming execution, with nondeterminism from the original run "influencing" the post-edit execution; and
- comparing the behavior of the original and altered runs to understand the effects of the speculative fix.

The nondeterministic input vectors for a JavaScript program are well-known and (compared to POSIX) very small in number [36]. However, defining post-edit replay semantics was previously an unsolved problem. Below, we define those semantics, describing how to execute post-edit code under the guidance of a log whose nondeterministic values may not cleanly apply to the post-edit execution. These post-edit replay semantics are an important contribution of the paper.

**Inside the call chain that contains the edit:** Once we have replayed execution to the edit point and modified the necessary state, we resume the call chain's execution. Post-edit, the chain may explore different branches than were visited in the original run. Thus, the chain may issue fewer or additional calls to nondeterministic functions like `Date()`.

- If the post-edit code makes *fewer* calls to a nondeterministic function $f$, we simply extract return values for $f$ from the log, replaying the same nondeterminism that the original run experienced. Once the call chain finishes, and we must replay the next event's call chain, we replay $f$'s values from the log, starting with the value that was first seen by the *original* execution of the call chain for the new event. For example, suppose that, during the original program execution, two events fired; the first call chain consumed random numbers $r_0 \ldots r_4$, and the second chain consumed $r_5 \ldots r_9$. Suppose that the first call chain is edited, such that it only makes two calls to `Math.random()`. When the second call chain executes in the post-edit run, `Math.random()` will return $r_5$, then $r_6$, and so on, since these are the random numbers that the second call chain saw during its original run.
- If the post-edit code generates *more* calls to a nondeterministic function than seen at logging time, we use a function-specific extrapolation technique to generate additional values once the call chain has exhausted the values that are associated with it in the log. For `Math.random()`, we simply generate new random numbers. For time-related functions like `Date()`, we return monotonically increasing time values that are smaller than the next logged time value. Once the call chain finishes and we trigger the call chain for a new event, we return to using the log to provide values for nondeterministic functions.

Post-edit code may also generate new externalized output. For example, an edited value may be written to local storage, or sent over the network via the query string of an `XMLHttpRequest`. Post-edit code may also modify event handler state in ways that cause fewer or additional events to fire in the future. For example, post-edit code may register timers that were never created in the original run; post-edit code may also *deregister* timers that fired in the original run. Post-edit code may also generate entirely new network requests, or register/deregister handlers for GUI events. Below, we discuss how to incorporate these changes into the post-edit universe.

**After the call chain which contains the edit has finished execution:** At this point, the replay framework has completed execution of the call chain. The framework can now manipulate program state before releasing the next event and invoking the appropriate event handlers.

Due to the edit, the current execution context may have different event handlers than what the program had at the equivalent moment in the original execution. The replay framework must integrate any changes into the log of nondeterminism; some post-edit events in the log must be marked as "do not replay," and some new events must be added to the log:

- If the edit resulted in the deletion of a timer, we mark all of the timer's subsequent events as "do not replay." If the edit created a new timer, we inject new timer events into the log, using the logged wall-clock time of preexisting events to determine where the new timer events should go, relative to the preexisting events.
- If the edit deleted a DOM handler, and the edited program has no remaining handlers for a particular event type, we mark all post-edit instances of that event as "do not replay." For example, if the deletion of a `keypress` handler leaves the program with no `keypress` handlers at all, we suppress future dispatches of logged `keypresses` (because such events cannot trigger

any call chains). If an edit registers a *new* DOM handler, then no special action is required—when the replay framework dispatches a relevant event, the framework will invoke the new handler as usual. Remember that Reverb records all GUI events at logging time, even if the application has not registered its own handlers for those events (§3.2). Thus, at replay time, Reverb can invoke new handlers for a particular event at the appropriate moment.

- If an edit closes `XMLHttpRequests` or `WebSockets`, the replay framework cancels future events that involve those network connections. If the edit creates a new, unlogged network request, then the replay framework must inject new network events into the log. If the server-side responder is also being replayed, then Reverb inserts a new request into the server-side log; the request represents a speculative server-side edit. When the response is generated, Reverb buffers it, and uses a model of network latency to determine where to inject the response into the client-side log (§3.4). If Reverb does not control the server-side responder, Reverb can terminate replay; alternatively, Reverb can issue the request to the live (but uncontrolled) responder, and then insert the response into a downstream position in the client log, using the observed network latency of the live fetch to determine where to place the response.
- The post-edit code may issue new reads or writes to local storage. The replay framework does nothing special to handle synchronous accesses to cookies or DOM storage—the framework simply passes those IOs to the underlying storage. For asynchronous accesses to `IndexedDB`, the replay framework must inject new IO events into the log, using a model for the expected latency of those events. Generating these events is logically similar to generating new network events, as explained in the previous bullet.

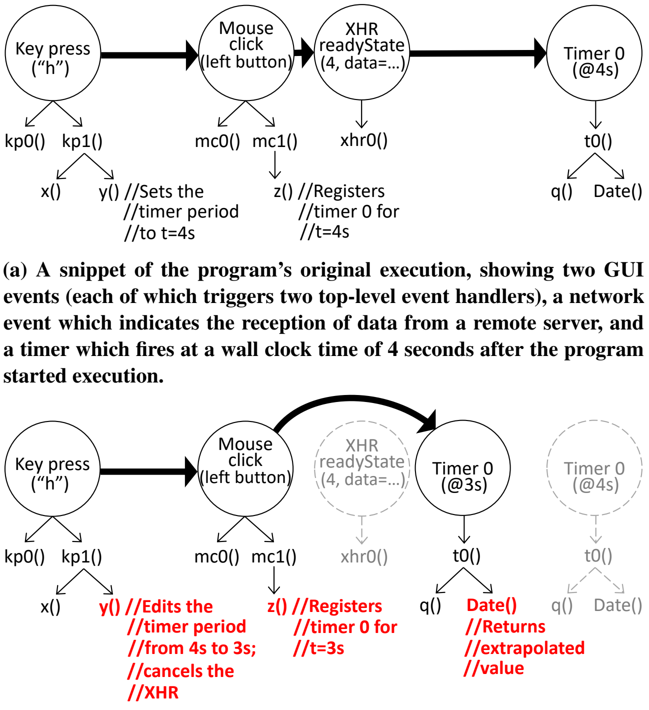
Note that the replay framework never injects new GUI events into the post-edit universe. For example, the framework will never inject new mouse clicks or key presses. Nothing prevents the framework from doing so, but, lacking a reasonable model for how user intent would change in the post-edit world, the framework is content to merely replay the GUI events from the original program run.

Once the replay framework has patched the log, the framework extracts the next high-level event from the log, and initiates the relevant call chain. The event may or may not have been seen in the original program run.

**Inside the call chain for a new event which did not occur during the original execution:** Replay uses extrapolation to generate return values for nondeterministic functions like `Math.random()`. When the call chain ends, we add and remove top-level events as described above.

**Inside the call chain for an event which *did* occur in the original execution:** We use the log to replay return values for nondeterministic functions; if the call chain's nondeterministic values are exhausted before the call chain finishes, we use extrapolation to generate additional values. When the call chain finishes, we add and remove events from the log as described above.

Figure 4 shows an end-to-end example of replaying events after an edit has been made. Once an altered replay finishes, developers can compare the data flows of the original and altered executions, looking for evidence that the hypothesized bug fix actually succeeded.


Key press ("h")
Mouse click (left button)
XHR readyState (4, data=...)
Timer 0 (@4s)
kp0()
kp1()
mc0()
mc1()
xhr0()
t0()
x()
y() //Sets the //timer period //to t=4s
z() //Registers //timer 0 for //t=4s
q()
Date()
Key press ("h")
Mouse click (left button)
XHR readyState (4, data=...)
Timer 0 (@3s)
Timer 0 (@4s)
kp0()
kp1()
mc0()
mc1()
xhr0()
t0()
t0()
x()
y() //Edits the //timer period //from 4s to 3s; //cancels the //XHR
z() //Registers //timer 0 for //t=3s
q()
Date() //Returns //extrapolated //value
q()
Date()

**(a) A snippet of the program's original execution, showing two GUI events (each of which triggers two top-level event handlers), a network event which indicates the reception of data from a remote server, and a timer which fires at a wall clock time of 4 seconds after the program started execution.**

**(b) During the replay process, the developer edits function `y()`. As a result, the XHR event is never replayed; additionally, the timer fires early, and receives a different value from `Date()`.**

**Figure 4: An example of how an edit changes the replay process. Beneath each event, we depict the associated call chains. Red indicates functions whose behavior is altered by the edit. Grey indicates events from the original execution which do not occur in the post-edit universe.**

Reverb uses classical string diffing algorithms [7, 11] to quickly identify the reads and writes that diverge in the two provenance chains. Reverb's logs contain enough information to reconstruct execution traces at the granularity of individual source code lines (see Figure 8); thus, developers can use differential slicing [28] to align divergent executions with respect to shared and non-shared lines of executed source code.

### 3.4 Debugging Across the Wide Area

**Node:** Node [56] is a server-side implementation of the JavaScript runtime. Like a browser-based JavaScript engine, Node exposes a single-threaded, event-driven interface. A Node application runs headless, i.e., without a GUI, but otherwise has access to timers, nondeterministic functions like `Date()`, and asynchronous IO channels like network sockets. Reverb interposes on these nondeterministic inputs using the same techniques that it leverages on the client-side.

To track causality between a client and a Node server, Reverb uses vector clocks [17, 33] to establish a partial ordering over the distributed events. At logging time, when a client issues an `XMLHttpRequest`, Reverb transparently adds a new cookie value which contains the client's clock. On the server-side, Reverb transparently modifies the HTTP request handler to extract the client clock and update the server's clock appropriately. When the server generates the HTTP response, Reverb uses a `Set-Cookie` header to transmit the server's updated clock to the client;[1] the client extracts the cookie and updates the local clock. The client browser automatically persists the cookie on local storage, as the browser would do for any other type of cookie.

In JavaScript, a program can associate a single top-level event with multiple handlers. At logging time, a client or server updates the local clock at the beginning of each event dispatch, before handlers run. The use of browser cookies to store client clocks allows a client to detect when *passively-fetched content* triggers server-side JavaScript execution. For example, suppose that client-side JavaScript code injects a new `<link>` tag into the page using the `innerHTML` DOM method; such a tag might represent a new style sheet. Client-side JavaScript will not have an opportunity to inspect the HTTP response headers for the `<link>`. However, when the next JavaScript-visible event fires, the first handler for that event can inspect the cookie that was set by the `<link>` fetch, extract the server's vector clock, and then update the local vector clock appropriately.

At replay time, Reverb collates the client logs and the server logs, using logical clocks to generate a total ordering over all events. Reverb then replays events from the total ordering; at each step, Reverb moves either a client or a server one event further in the global log.

Note that each host's log contains sufficient information to replay the host in isolation—the log contains all of the external nondeterministic stimuli that affected the host, as well as internal nondeterminism like GUI events or the values returned by clock reads. So, if a host communicates with multiple parties, but only some of them run Reverb, then the host can be replayed by itself, or in concert with some or all of the Reverb-enabled hosts. However, Reverb must be vigilant for speculative edits that generate new, unlogged requests to entities that are not participating in the replay (§3.3).

**Black-box components:** A client-side browser and a server-side Node engine both run single-threaded, event-driven JavaScript code. In contrast, server-side components like Redis and NGINX are single-threaded and event-driven, but are written in C, C++, or another non-JavaScript language. Reverb treats each such component as a black box, logging incoming requests and outgoing responses using a proxy. For example, our Reverb prototype intercepts HTTP traffic that is exchanged with a Redis server, using Redis-specific rules to extract `get(k)` and `put(k,v)` commands, and serialize the order in which commands are sent to Redis. Reverb assumes that each event handler inside a black box is deterministic, such that replaying a serialized stream of requests will result in 1) the same internal state for the component, and 2) the same responses being returned.[2] These assumptions are reasonable for server-side components like Redis that act as fairly simple front-ends to storage; however, these assumptions may not hold for server-side components that are written in arbitrarily-expressive, non-JavaScript languages like C++ or Go.

Reverb uses vector clocks to establish causality between black-box components and JavaScript-based components. However, Reverb does not log the reads and writes that black-box components make to internal state. Thus, data flows involving black-box state originate and terminate in the high-level requests and responses that black-box components exchange with external parties. For example, Reverb can track a JavaScript value on a client to the server-side Redis `get()` responses that influenced that value; however, Reverb cannot peer inside Redis to see why those responses were completed in a particular way.

At the beginning of logging, Reverb takes a snapshot of a black-box component's initial state using native mechanisms (e.g., Redis' built-in snapshot facility). At the beginning of replay, Reverb uses the snapshot to initialize the component.

**Speculative wide-area edits:** Reverb allows a developer to pause the wide-area application, edit client-side or server-side JavaScript, DOM, or storage state, and then resume execution. In general, Reverb uses the techniques from Section 3.3 to handle divergence, but wide-area debugging introduces some new divergence scenarios.

Define a requestor as a component that generates a request, and the responder as the component that responds. Browsers always acts as requestors, with server-side components acting as the corresponding responders; however, as server-side components talk to each other, they may act as requestors or responders at different points in time.

An edit may cause a responder to return a different response to a particular request, where "a particular request" is defined as a request made at a specific vector clock time. Reverb can detect such divergence because, at replay time, Reverb interposes on the methods that the responder uses to return data; Reverb compares the replay-time value of the response to the logged value from the original execution. If the values are different, Reverb rewrites the appropriate requestor-side log entries, propagating the new data. Later, replaying those events will naturally inject the new data into the requestor-side execution state. Subsequent requestor-side divergence is then handled using the approaches from Section 3.3.

If an edit induces a requestor to send a modified request to a responder, Reverb rewrites the appropriate responder-side log entry. When the replay logic applies the log entry to the responder, the replay logic buffers the response, and then replays the response on the requestor at the moment indicated in the log; note that the response may contain altered content with respect to the original, logged version of the response.

An edit can induce a requestor to generate a completely new request at a vector clock time that was not associated with a request during the original execution. In this scenario, Reverb's requestor-side replay driver does not allow the request to hit the real network. Instead, the responder-side driver injects a fake request into the responder-side log, and then resumes the replay process. Eventually, the replaying responder will handle the new event, and generate a response. The replay driver will buffer the response, and use a network model to determine when to replay the response on the requestor.

An edit may cause a requestor to not generate a logged request. In this case, Reverb does not replay the associated downstream events on the responder or the requestor. For example, if a browser

[1] A subtlety is that, if a server is concurrently handling multiple requests for a particular client, the server must ensure that the client receives a sequence of responses whose vector clocks have strictly increasing numbers in the server's slot. This policy is necessary because the client-side browser uses a "last-write-wins" policy for cookies.

[2] At replay time, individual responses may be emitted in a different order than the logging-time one, due to nondeterministic replay-time access delays to storage media like SSDs. However, Reverb's distributed replay driver buffers the responses, and ensures that response data is delivered to clients in the logging-time order.

does not issue a logged XMLHttpRequest to a Node server, then Reverb will not replay the Node-side HTTP request event, or the downstream browser-side events corresponding to the reception of the HTTP response.

**Replay subtleties:** A single web page can embed content from multiple origins. Cookies are isolated using the same-origin policy [44], so a page from origin foo.com cannot access cookies that are set by (say) <img> fetches to bar.com. Thus, JavaScript code in the enclosing foo.com page cannot read bar.com's latest vector clock. An in-browser implementation of Reverb can easily avoid this problem by allowing cross-origin cookie accesses when the browser is running in debug mode. A JavaScript-level implementation of Reverb must force all remote servers to reside in the same origin. This is often infeasible for the production version of a complex page, but possible for a testing version in which all page content is recorded using Mahimahi [52] or Fiddler [64], and then served from a single proxy that rewrites URLs to point to the proxy's origin.

A JavaScript-level implementation of Reverb must also be careful to replay load events properly. These events cannot be synthetically generated or deferred by JavaScript code, since JavaScript code has no ability to force the network stack to release bytes at controlled intervals. So, to properly replay the load event for a passively fetched object like an <img>, Reverb must ensure that, from a client's perspective, the <img> (and its vector-clock-containing cookie) arrive at a time that respects the vector clocks in the client-side log events. Practically speaking, this means that the server-side replay driver must coordinate with the client-side driver, and only release the last byte of a passively-fetched object when the wide-area replay has reached the appropriate point [36].

**Handling new features:** As with all existing systems for deterministic replay, Reverb must be updated as the components-to-log-and-replay are updated. We believe that determining when a new runtime interface or network command is added adds minimal burden to developers. For example, for Node [56], a Reverb developer can simply observe when the managed runtime gains new APIs, or loses old ones. Determining the appropriate semantics for speculative execution may be more difficult. However, we note that a key contribution of Reverb is in pointing out that, by analyzing a system at the level of managed code and single-threaded event loops, reasoning about speculative execution becomes easier.

# 4 IMPLEMENTATION

To log deterministic reads and writes to the JavaScript heap and the DOM, Reverb uses a modified version of Scout [51]. The stock version of Scout rewrites JavaScript and HTML, injecting instrumentation that runs during each read or write to JavaScript or DOM state. Reverb extends Scout so that it logs nondeterministic JavaScript events like mouse clicks and timer firings. At replay time, Reverb reconstructs data flows using the low-level Scout traces. Reverb defines a default set of data flow manipulations, like targeted dynamic traces (§3.2). However, Reverb stores raw data flow logs in a simple JSON format, and defines a plugin model which allows developers to create their own queries. To display data flow graphs, Reverb uses the NEATO visualization library [22].

At replay time, Reverb injects a custom JavaScript library into the application code that runs on a browser or a Node instance. The library acts as a replay driver, dispatching high-level events from the log as requested by the human developer who is managing the debugging workflow. The event dispatch process is similar to that of prior replay frameworks like Mugshot [36] or Jardis [8], although Reverb dispatches events across multiple hosts during wide-area replay (§3.4). Black-box components like Redis are logged and replayed using a component-specific replay proxy (§3.4).

During replay, Reverb uses Mahimahi [52], a record-and-replay framework for HTTP requests, to serve browser-side content that is fetched via the src attribute of HTML tags. Content that is actively fetched by JavaScript code is served by the replay driver, from the log of nondeterministic events. During a wide-area replay that involves clients and server-side components, Mahimahi only returns passively-fetched content that was not returned by a server-side component during the original execution.

To support speculative edit-and-continue, Reverb must be able to modify the code or data belonging to a paused JavaScript execution context. One implementation option would be to change the C++ code inside a JavaScript engine to expose mutation hooks for internal state. Our Reverb prototype uses a different approach—it executes the JavaScript code atop MetaES [9], a JavaScript interpreter that is written in JavaScript. This approach allows Reverb to be used with arbitrary client browsers or Node implementations, since Reverb can mutate application state without assistance from the underlying JavaScript engine. A developer expresses a pause point as a 2-tuple consisting of a source code line and a trigger condition, e.g., "the i-th iteration of the enclosing loop." Reverb will pause the MetaES interpreter at the appropriate moment. The developer can inspect the program state, devise an edit, and then express that edit to Reverb in the form of a JavaScript statement for Reverb to speculatively apply to the replay (§3.3).

# 5 EVALUATION

In this section, we demonstrate that Reverb is an *efficient, helpful tool for bug analysis*.

## 5.1 The Tractability of Data Flow Analysis

Intuition might suggest that tracking all deterministic and nondeterministic events would produce huge logs. However, in the Alexa Top 300 pages [3], the median number of reads and writes that occur during a page load are 13,275 and 6,328, respectively. Those numbers are surprisingly low, given the fact that an average web page includes 401 KB of JavaScript source code [68]. However, diagnosing bugs is still tricky: a graph with thousands of nodes is small enough to be efficiently analyzed by a computer, but big enough to be hard for a human to understand. For example, across the 300 test pages:

- the median number of writes per variable was 8, with a 95th percentile of 210;
- the median number of unique source code lines that wrote a variable was 5, with a 95th percentile of 22;
- when considering the final value for each variable, the median length of the value's provenance chain (§3.2) was 16, with a 95th percentile of 131.

These statistics are for the JavaScript code which executes during a page load. After the load completes, additional JavaScript executes in response to GUI interactions, the firing of timers, and so on. Executing post-load call chains results in more reads and writes for

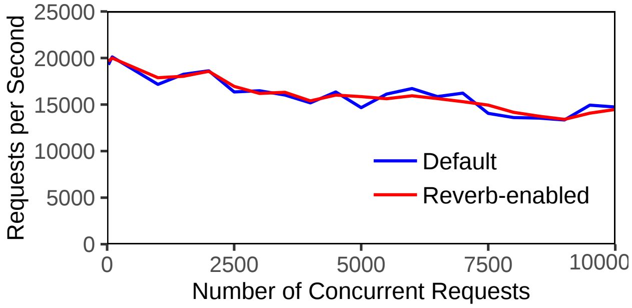


**Figure 5: Response throughput for two versions of a Node server. Each data point represents the throughput across 100,000 requests.**

Reverb to track, but the volume is low compared to the activity that is generated by the initial page load. As a concrete example, on the `wsj.com` website, hovering over a menu item at the top of the page will trigger several event handlers for mouse activity. However, firing these handlers only generates 486 reads and 107 writes. In contrast, the initial page load generates 33,844 reads and 16,121 writes.

Using Mahimahi [52], we loaded Reverb-instrumented pages under a variety of emulated network conditions, measuring the client-perceived impact of Reverb's instrumentation. Due to space restrictions, we elide a full discussion of the results, and just note that Reverb's client-side instrumentation is fast enough to add to real, customer-facing pages; for example, on a 12 Mbits/s link with a 50 ms RTT, median page load time slows by just 5.5%. The logs for Reverb-instrumented pages also grow slowly: across our 300 page corpus, the median (gzipped) log size after a page load was 45.4 KB, with a 95th percentile size of 113.2 KB. Given such a log, Reverb required a median of 7.8 seconds to generate a full data flow graph; the 95th percentile time was 32.3 seconds. Note that graph generation can be performed in the background during the logging run. Thus, much or all of the cost can be paid before a human developer begins the debugging process.

### 5.2 Server-side Overheads

Reverb's logging approach for a Node server is similar to Reverb's logging approach for a client-side browser (§3.4). However, a Node server that handles many clients will produce log entries more quickly than a client browser which loads a single page and then intermittently handles user input. To examine Reverb overheads on Node, we wrote a simple Node web server. For each request, the server returned the dynamic string `"Hello world at " + (new Date()).getTime()`. For each request, Reverb had to log the incoming HTTP request, a few dozen reads and writes inside the server's request handler, the timestamp returned by `Date()`, and the outgoing HTTP response. This toy server was a pessimistic test of Reverb's overheads, since real server code has a higher ratio of executed source code lines per nondeterministic value logged.

We used the Apache benchmarking tool `ab` [6] to generate HTTP requests. We placed the Node server and `ab` on the same machine, to emphasize Reverb's computational overheads. As shown in Figure 5, we varied the number of concurrent client requests from 25 to 10,000, measuring response throughput for a normal version of the server, and a Reverb-enabled variant. The throughputs of the two servers were within 3% of each other. CPU utilization was also similar for the two servers.

| | Mailpile | EtherCalc |
|---|---|---|
| Total writes | 24,202 | 31,251 |
| Total reads | 67,335 | 89,737 |
| JavaScript heap objects | 2,822 | 4,028 |
| DOM nodes | 619 | 1,531 |
| Wall-clock time to bug at logging time | 12.8 secs | 10.4 secs |
| Wall-clock time to bug at replay time | 3.1 secs | 3.9 secs |

**Figure 6: Summary statistics for the Mailpile and EtherCalc case studies. Note that, during replay, Reverb can skip user think time, so Reverb can replay a buggy execution faster than it originally occurred.**

The growth of Reverb's compressed log was 258 Kbps (equivalent to 32.3 KB per second). Note that black box components like Redis have slower log growth—for these components, Reverb logs incoming requests and responses, but not deterministic reads or writes to internal black-box state.

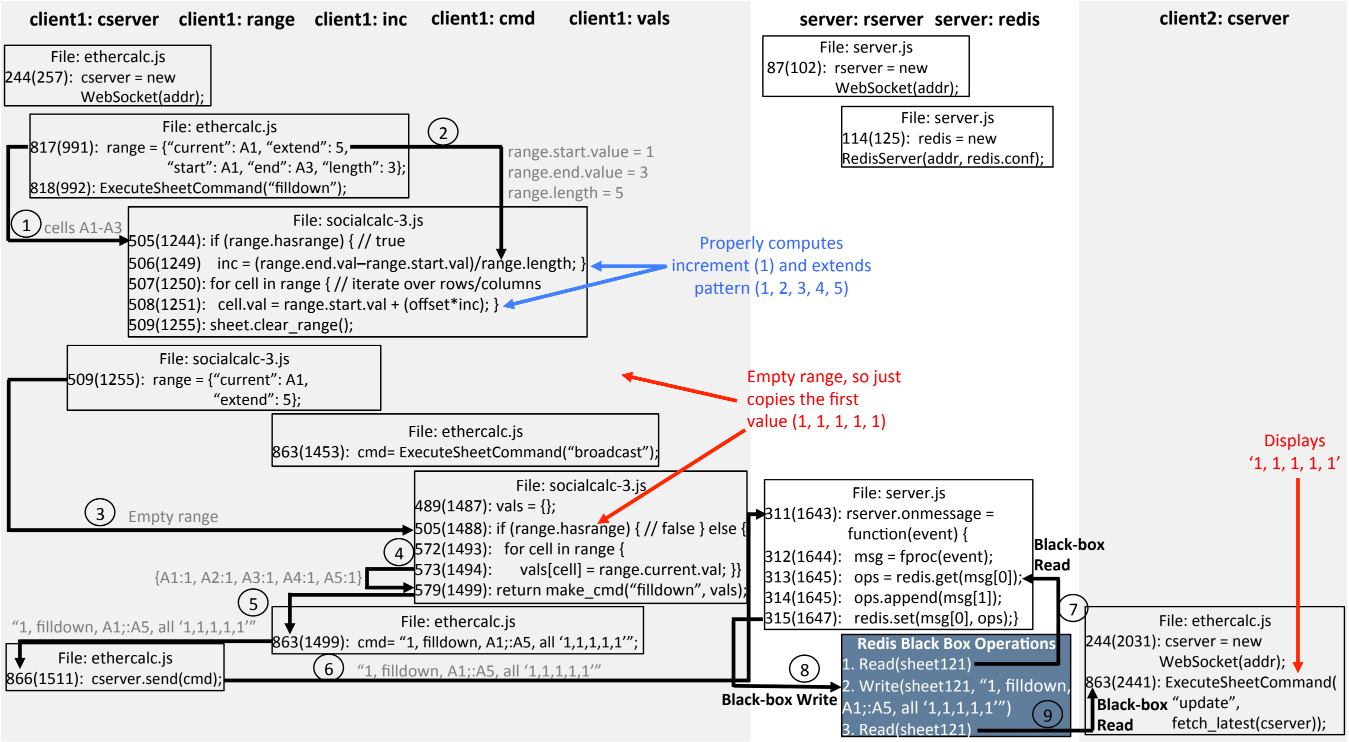
### 5.3 Bug Diagnosis Case Study: EtherCalc

Evaluating a new debugging platform is tricky, and partially subjective. In this section, we provide an in-depth case study of how we used Reverb to debug a web application that we did not create, and whose code we had no previous familiarity with.

As shown in Figure 1, EtherCalc is a collaborative spreadsheet application. A single document can be simultaneously viewed and edited by multiple users, with a Node server disseminating updates across browsers, and storing the canonical spreadsheet state in a Redis database. EtherCalc is the largest application that we examined, consisting of 36 HTML files, 899 JavaScript files, and 232,662 total lines of code. Figure 6 provides additional statistics about the application.

Bug #314 involves a broken propagation of auto-fill operations between two browsers. In an auto-fill operation, a user enters data (e.g., "1,2,3") into a few exemplar cells; the user then highlights the cells, and drags the bottom edge of the highlighted region downward, causing the spreadsheet to guess the pattern in the exemplar cells and automatically apply the pattern (e.g., "4,5") to subsequent cells. In Bug #314, auto-fill operations that are generated on one browser are not properly delivered to other browsers. In the example above, the first browser (`client1`) would correctly display "1,2,3,4,5", but the second browser (`client2`) would display "1,1,1,1,1". We recreated this problem, recording a buggy session that involved two browsers, a Node server, and a Redis server.

**Diagnosing the bug:** Figure 7 annotates a targeted dynamic trace for the buggy execution; to make room for the annotations, we removed the more obvious data flows. When `client1`'s user initiates an auto-fill, EtherCalc creates a `range` object which describes the auto-fill operation; for example, the `range` object contains the start cell and ending cell for the base pattern, as well as the start cell and ending cell for the range where the extrapolated data should be placed. EtherCalc then calls `ExecuteSheetCommand()`, using the `filldown` parameter to indicate a pattern extension request. The function checks whether a valid `range` object is present (1), and if it exists, the function uses values in the object to determine



```
File: ethercalc.js
244(257):  cserver = new
           WebSocket(addr);
```

```
File: ethercalc.js
817(991):  range = {“current”: A1, “extend”: 5,
                    “start”: A1, “end”: A3, “length”: 3};
818(992): ExecuteSheetCommand(“filldown”);
```

```
File: socialcalc-3.js
505(1244): if (range.hasrange) { // true
506(1249)    inc = (range.end.val–range.start.val)/range.length; }
507(1250): for cell in range { // iterate over rows/columns
508(1251):   cell.val = range.start.val + (offset*inc); }
509(1255): sheet.clear_range();
```

```
File: socialcalc-3.js
509(1255):  range = {“current”: A1,
                     “extend”: 5};
```

```
File: ethercalc.js
863(1453):  cmd= ExecuteSheetCommand(“broadcast”);
```

```
File: socialcalc-3.js
489(1487): vals = {};
505(1488): if (range.hasrange) { // false } else {
572(1493):   for cell in range {
573(1494):       vals[cell] = range.current.val; }}
579(1499): return make_cmd(“filldown”, vals);
```

```
File: ethercalc.js
863(1499):  cmd= “1, filldown, A1;:A5, all ‘1,1,1,1,1’”;
```

```
File: ethercalc.js
866(1511):  cserver.send(cmd);
```

```
File: server.js
87(102):  rserver = new
          WebSocket(addr);
```

```
File: server.js
114(125):  redis = new
RedisServer(addr, redis.conf);
```

```
File: server.js
311(1643): rserver.onmessage =
               function(event) {
312(1644):   msg = fproc(event);
313(1645):   ops = redis.get(msg[0]);
314(1645):   ops.append(msg[1]);
315(1647):   redis.set(msg[0], ops);}
```

```
Redis Black Box Operations
1. Read(sheet121)
2. Write(sheet121, “1, filldown,
A1;:A5, all ‘1,1,1,1,1’”)
3. Read(sheet121)
```

```
File: ethercalc.js
244(2031): cserver = new
              WebSocket(addr);
863(2441): ExecuteSheetCommand(
              “update”,
              fetch_latest(cserver));
```

**Figure 7: A buggy EtherCalc session. The *i*-th executed source code line is prefixed with `lineNumInSrcFile(i)`. Black arrows represent data flows between executed source code lines: data written by the source code at the base of an arrow is read by source code at the head of the arrow. The blue and red text was manually added to highlight specific parts of the debugging narrative; Reverb generates the information in the rest of the diagram.**

the appropriate increment value for the pattern extension (2). After applying the auto-fill to `client1`'s GUI and JavaScript state, `ExecuteSheetCommand()` overwrites `client1`'s `range` object, effectively removing several properties like `start` and `extend`. `client1` then calls `ExecuteSheetCommand()` again, passing a different parameter called `broadcast`. This parameter specifies that `ExecuteSheetCommand()` should *not* apply the operation to local state, but instead send that operation to the Node server, who will persist the operation in Redis and then distribute the operation to other clients. As in the first execution of `ExecuteSheetCommand()`, the function checks whether a valid `range` object exists. However, in this case, because the prior function execution partially cleared the `range` object, the validity check fails (3). As a result, `ExecuteSheetCommand()` simply generates an operation which copies the value in `range.current` to all cells covered by `range.current` to `range.length` (4); note that these two properties were not removed during the earlier reset of the `range` object. The resulting operation (5) is what will eventually create the autofill “1, 1, 1, 1, 1” (rather than “1, 2, 3, 4, 5”) on `client2`. `client1` sends the operation to the Node server (6), triggering a server-side `onmessage` handler. The handler issues a black-box read to Redis, fetching the list of all operations that have been applied to the spreadsheet (7). The handler appends the new operation to the list, and then uses a black-box write to store the updated list on Redis (8). Later, when `client2` opens the spreadsheet (9), `client2` will fetch the buggy operation list and use the list to update the local GUI.

**Fixing the bug:** Reverb made it easy for us to generate a wide-area data flow graph. However, these graphs do not automatically provide a bug fix; instead, the graphs help to localize where buggy state is being created, and how that state is being propagated. Thus, our next task was to try to actually fix the bug. An obvious potential fix was to modify `ExecuteSheetCommand()` so that the first invocation did not reset the `range` object. We made this edit and then performed a speculative replay. The edit initially appeared successful—the Node server received the correct auto-fill operation, stored it on Redis, and then sent it to the second browser, who correctly applied the operation. However, our recorded session contained *two* auto-fill operations involving two distinct sets of cells; our hypothesized fix prevented the second auto-fill operation from appearing in the GUI of the first browser or the second browser. Looking at the distributed data flow, we saw that the first browser was not generating a second auto-fill locally (and therefore was not sending a second auto-fill operation to the Node server).

Further investigation of `ExecuteSheetCommand()`'s code revealed that the first call in a pair of invocations expects the `range` object to be set to a default value—otherwise, `ExecuteSheetCommand()` terminates without updating the local spreadsheet or sending an update to remote clients via the Node server. We tried a different bug fix in which the second call to `ExecuteSheetCommand()` clears the `range` object at the *end* of `ExecuteSheetCommand()`'s execution. The speculative replay for this fix led to no problems—both auto-fill operations were properly displayed on both browsers.

Of course, the successful speculative replay was not a proof of the fix's correctness; the successful replay was essentially the successful passing of a unit test involving a particular usage scenario. However, this case study demonstrates how replay debugging, wide-area data flow tracking, and speculative edits work in concert to ease the cognitive overhead of understanding large code bases.

### 5.4 Additional Case Studies

We have used Reverb to diagnose a variety of additional bugs. The appendix (§A.1 and §A.2) of the extended technical report [5] provides detailed case studies for two of them. The first case study demonstrates Reverb's usefulness in debugging visual errors involving DOM misconfiguration; the second case study shows how Reverb helps developers to fix errors in third-party JavaScript code that local developers did not write.

### 5.5 Speculatively Replaying Known-Good Bug Fixes

As another test of Reverb's efficacy, from the public bug database of jQuery [65], we examined five random bugs from the past eight years, which had successfully been resolved, and whose bug tracker descriptions were sufficiently detailed for us to recreate the bug.[3] jQuery is a popular client-side JavaScript library for DOM manipulation; the library consists of roughly 6,600 lines of code. For each of the resolved bugs, we did the following:

- First, we downloaded the buggy version of jQuery that immediately preceded the patched version.
- We verified that Reverb could reproduce the bug using traditional (i.e., non-speculative) deterministic replay.
- We then replayed the library to the moment that preceded the faulty behavior. We paused the replay at that point, applied the known-good bug fix from the bug database, and then resumed the now-speculative replay to see whether the replay would finish without displaying the faulty behavior.

For all five cases, Reverb's edited replay correctly indicated that the "speculative" bug fix was indeed a correct one.

### 5.6 User Study

To determine whether Reverb would be useful to people besides the paper authors, we performed a small user study involving six participants. All participants had prior experience with front-end web development, and all possessed at least some familiarity with traditional in-browser debuggers. All participants brought their own laptops to the user study, but debugging exercises were performed inside of a VM provided by the paper authors, to ensure uniformity of experience.

For each user, we first provided a refresher tutorial about how the traditional debugger works; we discussed topics like watchpoints, breakpoints, and code prettification. We also explained how Reverb's data flow graphs are generated and refined via human-driven queries. Once both tutorials were complete, we presented the user with a real bug to diagnose. The bug involved a JavaScript exception being thrown by a real, complex web page; refer to Section A.2 in the extended technical report [5] for a detailed description of the bug. We divided the participants equally, creating two groups of three; the first group was asked to use Reverb to diagnose the bug, whereas the second group was asked to use the traditional in-browser debugger. Each person was given ten minutes to diagnose the bug. When a user believed that she had found the root cause for the bug, she informed the paper authors, who then verified whether the hypothesized root cause was correct. If it was not, the user was told to keep working until a correct diagnosis was generated, or ten minutes elapsed.

Two of the Reverb users diagnosed the bug in less than five minutes, with the third user did so within ten minutes. In contrast, one user of the traditional debugger failed to diagnose the error within the ten minute window. The remaining two users of the traditional debugger correctly diagnosed the fault before the timer elapsed, but required more than five minutes. Thus, these results suggest that Reverb is a more powerful diagnostic tool than a traditional debugger.

At the end of the study, we asked the participants some qualitative questions. In response to the question "Would you prefer to use Reverb over a traditional debugger?", five out of six participants said yes. The one person who disagreed complained about Reverb's GUI (which is admittedly rudimentary in our prototype). In particular, the complaining user said that graph querying and pruning was unnecessarily awkward. When asked "Would Reverb-style data flow operations be a useful compliment to standard debugging primitives?", all six participants said yes. Furthermore, all three Reverb users declared, without prompting, that speculative edit-and-continue would be a powerful debugging feature.

Testing bug fixes is inherently unsound, even without Reverb (§1); unsurprisingly, the replay of an edited execution may occasionally lead to confusing results. In the authors' own experience, these incidents usually involve the replaying of GUI events. For example, if an edit changes the visual locations of DOM nodes, then replaying (say) a mouse click to a particular $(x, y)$ coordinate may result in unexpected event handlers firing. Reverb's ability to diff the data flows and control flows of a logged execution and an edited one can identify the reason for divergence, but developers must be diligent about checking the diffs when exploring counterintuitive post-edit behaviors.

## 6 CONCLUSION

Reverb is the first replay debugger that tracks fine-grained, wide-area data flows while also supporting speculative edit-and-continue. Such capabilities were impossible in prior debugging frameworks because those frameworks logged program behavior at the wrong level of abstraction; in contrast, Reverb efficiently tracks behavior at the level of managed code and single-threaded event loops. Case studies demonstrate that Reverb is a powerful tool for diagnosing real, complex bugs.

## REFERENCES

[1] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. 1991. Dynamic Slicing in the Presence of Unconstrained Pointers. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*. ACM.

[3] We examined bugs #3439, #3472, #3571, #3573, and #3579.

[2] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI)*. ACM.

[3] Alexa. [n.d.]. Top Sites in the United States. http://www.alexa.com/topsites/countries/US.

[4] Bowen Alpern, Ton Ngo, Jong-Deok Choi, and Manu Sridharan. 2000. DejaVu: Deterministic Java Replay Debugger for JalapeÑO Java Virtual Machine. In *Proceedings of OOPSLA*. ACM.

[5] Anonymous. [n.d.]. Reverb Technical Report. https://github.com/anonymous-tr/Reverb_SOCC19_Technical_Report/blob/master/reverb_technical_report.pdf.

[6] Apache Software Foundation. 2017. https://httpd.apache.org/docs/2.4/programs/ab.html.

[7] Gregory V. Bard. 2007. Spelling-error Tolerant, Order-independent Pass-phrases via the Damerau-levenshtein String-edit Distance Metric. In *Proceedings of the Fifth Australasian Symposium on ACSW Frontiers (ACSW)*. Australian Computer Society, Inc.

[8] Earl T. Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. 2016. Time-Travel Debugging for JavaScript/Node.js. In *Proceedings of the 2016 International Symposium on the Foundations of Software Engineering (FSE)*. ACM.

[9] Bartosz Krupa. [n.d.]. MetaES. https://github.com/metaes/metaes.

[10] Jean-Francois Bergeretti and Bernard A. Carré. 1985. Information-flow and Data-flow Analysis of While-programs. *ACM Trans. Program. Lang. Syst.* 7, 1 (Jan. 1985).

[11] Philip Bille. 2005. A Survey on Tree Edit Distance and Related Problems. *Theor. Comput. Sci.* 337, 1-3 (June 2005), 217–239.

[12] David Binkley, Mark Harman, and Jens Krinke. 2007. Empirical Study of Optimization Techniques for Massive Slicing. *ACM Trans. Program. Lang. Syst.* 30, 1, Article 3 (November 2007).

[13] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. 2002. ReVirt: Enabling Intrusion Analysis Through Virtual-machine Logging and Replay. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002).

[14] Eclipse. 2016. FAQ What is hot code replace? https://goo.gl/brp5oQ.

[15] Stuart I. Feldman and Channing B. Brown. 1988. IGOR: A System for Program Debugging via Reversible Execution. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (PADD)*. ACM.

[16] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349.

[17] Colin J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference* 10 (1988), 56–66.

[18] Firebug. [n.d.]. JavaScript Debugger and Profiler. http://getfirebug.com/javascript.

[19] GDB: The GNU Project Debugger. 2012. GDB and Reverse Debugging. https://www.gnu.org/software/gdb/news/reversible.html.

[20] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. 2006. Replay Debugging for Distributed Applications. In *Proceedings of the USENIX Annual Technical Conference (ATEC)*. USENIX.

[21] Google Developers. [n.d.]. Debug. https://developers.google.com/web/tools/chrome-devtools/debug/?hl=en.

[22] Graphviz. [n.d.]. Drawing Graphs with NEATO. http://www.graphviz.org/pdf/neatoguide.pdf.

[23] Tibor Gyimóthy, Árpád Beszédes, and Istán Forgács. 1999. An Efficient Relevant Slicing Method for Debugging. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7)*. Springer-Verlag, London, UK, UK.

[24] Christian Hammer and Gregor Snelting. 2009. Flow-sensitive, Context-sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs. *Int. J. Inf. Secur.* 8, 6 (Oct. 2009).

[25] Ian Hickson. 2015. Web Workers. https://www.w3.org/TR/workers/.

[26] Shin Hong, Yongbae Park, and Moonzoo Kim. 2014. Detecting Concurrency Errors in Client-Side Java Script Web Applications. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation (ICST)*. IEEE Computer Society.

[27] Jon Howell, Bryan Parno, and John R. Douceur. 2013. Embassies: Radically Refactoring the Web. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX.

[28] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. 2011. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society.

[29] Andrew J. Ko and Brad A. Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. ACM.

[30] Bogdan Korel and JJanusz Laski. 1988. Dynamic Program Slicing. *Inform. Process. Lett.* 29, 3 (Oct. 1988).

[31] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. 2008. Practical Object-Oriented Back-in-Time Debugging. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming (ECOOP)*. Berlin, Heidelberg.

[32] Mailpile. 2017. https://github.com/mailpile/Mailpile.

[33] Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel & Distributed Algorithms*, M. Cosnard et. al. (Ed.). Elsevier Science Publishers B. V., 215–226.

[34] James Mickens. 2012. Rivet: Browser-agnostic Remote Debugging for Web Applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association.

[35] James Mickens and Mohan Dhawan. 2011. Atlantis: Robust, Extensible Execution Environments for Web Applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*. ACM.

[36] James Mickens, Jeremy Elson, and Jon Howell. 2010. Mugshot: Deterministic Capture and Replay for Javascript Applications. In *Proceedings of NSDI*.

[37] Microsoft. [n.d.]. Edit and Continue. https://msdn.microsoft.com/en-us/library/bcew296c.aspx.

[38] Mozilla Developer Network. [n.d.]. Concurrent model and Event Loop. goo.gl/UmzCa5.

[39] Mozilla Developer Network. [n.d.]. Debugger. https://developer.mozilla.org/en-US/docs/Tools/Debugger.

[40] Mozilla Developer Network. 2016. Document Object Model (DOM). https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model.

[41] Mozilla Developer Network. 2016. HTTP Cookies. https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies.

[42] Mozilla Developer Network. 2016. <iframe>. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe.

[43] Mozilla Developer Network. 2016. IndexedDB API. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API.

[44] Mozilla Developer Network. 2016. Same-origin Policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[45] Mozilla Developer Network. 2016. Web Storage API. https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API.

[46] Mozilla Developer Network. 2016. WebSocket. https://developer.mozilla.org/en-US/docs/Web/API/WebSocket.

[47] Mozilla Developer Network. 2016. XMLHttpRequest. https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest.

[48] Kiran-Kumar Muniswamy-Reddy, David Holland, Uri Braun, and Margo Seltzer. 2006. Provenance-Aware Storage Systems. In *Proceedings of USENIX ATC*.

[49] Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. 2015. Detecting JavaScript Races That Matter. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM.

[50] Nicholas Nethercote and Alan Mycroft. 2003. Redux: A dynamic dataflow tracer. In *Electronic Notes in Theoretical Computer Science*. Elsevier.

[51] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *Proceedings of NSDI*. USENIX Association.

[52] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proceedings of ATC*. USENIX Association, Santa Clara, CA.

[53] NGINX Inc. 2017. NGINX. https://www.nginx.com/.

[54] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2014. Building Call Graphs for Embedded Client-side Code in Dynamic Web Applications. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM.

[55] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2015. Cross-language Program Slicing for Dynamic Web Applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM.

[56] Node.js Foundation. 2017. Node.js. https://nodejs.org/en/.

[57] Frolin S. Ocariza, Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2013. An Empirical Study of Client-Side JavaScript Bugs. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*.

[58] Frolin S. Ocariza, Guanpeng Li, Karthik Pattabiraman, and Ali Mesbah. 2016. Automatic Fault Localization for Client-side JavaScript. *Softw. Test. Verif. Reliab.* 26, 1 (Jan. 2016).

[59] Frolin S. Ocariza, Karthik Pattabiraman, and Benjamin Zorn. 2011. JavaScript Errors in the Wild: An Empirical Study. In *Proceedings of the 2011 IEEE 22Nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society.

[60] Redis Labs. 2017. Redis. https://redis.io/.

[61] Sam Shah, Craig Soules, Greg Ganger, and Brian Noble. 2007. Using Provenance to Aid in Personal File Search. In *Proceedings of USENIX ATC*.

[62] Matthew Tancreti, Vinaitheerthan Sundaram, Saurabh Bagchi, and Patrick Eugster. 2015. TARDIS: Software-only System-level Record and Replay in Wireless Sensor Networks. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks (IPSN)*. ACM.

[63] Audrey Tang. 2017. EtherCalc: A Web Spreadsheet. https://ethercalc.net/.

[64] Telerik. [n.d.]. Fiddler. http://www.telerik.com/fiddler.

[65] The jQuery Foundation. 2018. jQuery. https://jquery.com/.

[66] Frank Tip. 1994. *A Survey of Program Slicing Techniques.* Technical Report. CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands.

[67] Nicolas Viennot, Siddharth Nair, and Jason Nieh. 2013. Transparent Mutable Replay for Multicore Debugging and Patch Validation. In *Proceedings of ASPLOS*.

[68] WebPagetest. 2016. HTTP Archive - Interesting Stats. https://goo.gl/9V4KJn.

[69] Mark Weiser. 1982. Programmers Use Slices when Debugging. *Commun. ACM* 25, 7 (July 1982), 446–452.

[70] Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2017. Automated Bug Removal for Software-Defined Networks. In *Proceedings of NSDI*.

[71] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2014. Diagnosing Missing Events in Distributed Systems with Negative Provenance. In *Proceedings of SIGCOMM*.

[72] Yunhui Zheng, Tao Bao, and Xiangyu Zhang. 2011. Statically Locating Web Application Bugs Caused by Asynchronous Calls. In *Proceedings of the 20th International Conference on World Wide Web (WWW)*. ACM.

[73] Wenchao Zhou, Qiong Fei, Narayan Arjun, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. 2011. Secure Network Provenance. In *Proceedings of SOSP*.

[74] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. 2010. Efficient Querying and Maintenance of Network Provenance at Internet-Scale. In *Proceedings of SIGMOD*.