Sinatra: Stateful Instantaneous Updates for Commercial Browsers Through Multi-Version eXecution

Ugnius Rumsevicius
University of Illinois at Chicago, IL, USA

Siddhanth Venkateshwaran
University of Illinois at Chicago, IL, USA

Ellen Kidane
University of Illinois at Chicago, IL, USA

Luís Pina
University of Illinois at Chicago, IL, USA

Abstract

Browsers are the main way in which most users experience the internet, which makes them a prime target for malicious entities. The best defense for the common user is to keep their browser always up-to-date, installing updates as soon as they are available. Unfortunately, updating a browser is disruptive as it results in loss of user state. Even though modern browsers reopen all pages (tabs) after an update to minimize inconvenience, this approach still loses all local user state in each page (e.g., contents of unsubmitted forms, including associated JavaScript validation state) and assumes that pages can be refreshed and result in the same contents. We believe this is an important barrier that keeps users from updating their browsers as frequently as possible.

In this paper, we present the design, implementation, and evaluation of Sinatra, which supports instantaneous browser updates that do not result in any data loss through a novel Multi-Version eXecution (MVX) approach for JavaScript programs, combined with a sophisticated proxy. Sinatra works in pure JavaScript, does not require any browser support, thus works on closed-source browsers, and requires trivial changes to each target page, that can be automated. First, Sinatra captures all the non-determinism available to a JavaScript program (e.g., event handlers executed, expired timers, invocations of `Math.random`). Our evaluation shows that Sinatra requires 6MB to store such events, and the memory grows at a modest rate of 253KB/s as the user keeps interacting with each page. When an update becomes available, Sinatra transfer the state by re-executing the same set of non-deterministic events on the new browser. During this time, which can be as long as 1.5 seconds, Sinatra uses MVX to allow the user to keep interacting with the old browser. Finally, Sinatra changes the roles in less than 10ms, and the user starts interacting with the new browser, effectively performing a browser update with zero downtime and no loss of state.

2012 ACM Subject Classification Computer systems organization → Availability; Software and its engineering → Maintaining software

Keywords and phrases Internet browsers, dynamic software updating, multi-version execution

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.26

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.9.2.3`

Funding This work was funded in part by NSF CCF-2227183.

1 Introduction

Browsers are the main way in which most users experience the internet. Browsers are responsible for the safety of user sensitive data, in the form of cookies, saved passwords, and credit card information, and other personal information used to auto-complete forms.

Browsers are also responsible for ensuring the integrity of the websites that the user visits, checking certificates and negotiating encrypted HTTPS channels. Given all this, browsers are prime targets for malicious entities. For the common user, the best way to protect their browsers (and the personal data they keep) is to keep the browsers as up-to-date as possible.

Unfortunately, users are slow to update their browser to a new version. In terms of percentage of users, data for Google Chrome [53] and Mozilla Firefox [41] show that a new browser version takes about 2 weeks to overtake the previous version, and about 4 weeks to reach its peak. Given the fast pace of browser releases (6 weeks for Google Chrome and 4 weeks for Mozilla Firefox), the amount of users running outdated versions is significant at any given time.

Browser developers are aware of the problems caused by running outdated versions, and provide features to entice users to update, from reminding the user that a new update is available to minimizing the inconvenience by reopening all pages (tabs) after the update. Even though popular, the latter feature has three main flaws. **First**, it disrupts the user interaction by closing the browser, downloading the new version, and then opening it. **Second**, it assumes that pages can simply be refreshed after the update. Such an assumption fails if a login session expires, which causes the page to refresh to the login portal; or if the contents of the page change with each refresh, as is the case with modern social media. **Third**, refreshing a page loses all user state accumulated on that page since it was loaded. Such state includes, among others, data in HTML forms and JavaScript state.

The result is simple: **Browser updates are disruptive for the average user**. Dynamic Software Updating (DSU) techniques can be used for eliminating such disruption, updating a program in-process. Unfortunately, state-of-the-art DSU tools cannot handle programs as complex as modern commercial internet browsers (Section 2.1). Also, simply dumping the old browser memory state to disk and reloading it in the new browser does not work, as the new browser may change the internal state representation.

In this paper, we present the design and implementation of **Sinatra**– **Stateful Instantaneous browser updates** – a novel MVX technique implemented in pure JavaScript. SINATRA requires little changes to the target JavaScript application (Section 3), which can be performed automatically for all the pages accessed through an HTTP proxy (Section 3.1).

To perform an update (Section 3.2), SINATRA captures all sources of non-determinism accessed by the browser. Then, when an update becomes available, SINATRA launches the updated browser as a separate process, and feeds it the same non-determinism, thus synchronizing the JavaScript state between both browsers. During this time, SINATRA allows the user to keep interacting with the old browser by performing MVX until the updated browser's state is up-to-date. Once the update was successful, SINATRA terminates the old browser and the user can start interacting with the new browser.

Note that simply transferring the JavaScript between browsers is not sufficient for two reasons. First, the user cannot interact with either browser while transferring the state. Second, failed updates may still result in loss of user data. Multi-Version eXecution (MVX) solves both problems by allowing the user to interact with the old browser while the new browser is receiving the state, and by allowing SINATRA to cancel a failed update simply by closing the new browser. Unfortunately, state-of-the-art MVX tools cannot handle modern commercial internet browsers (Section 2.2), and performing MVX at the JavaScript is not as straightforward due to the event-driven programming paradigm (Section 2.3).

SINATRA captures all sources of non-determinism available to a JavaScript program, including execution of event handlers (Sections 3.3), and non-deterministic functions such as `Math.random` (Section 3.4). We implemented SINATRA in pure JavaScript using an extra *coordinator* process to enable communication between browsers (Section 4.1) that serializes JavaScript non-determinism as JSON (Section 4.2). Our implementation also handles all other sources of state in a JavaScript application (Sections 4.2 and 4.3).

This paper also presents an extensive evaluation of SINATRA using 4 JavaScript applications and realistic workloads (Section 6.1) and 2 real-world modern websites (Section 6.8). Our results show that SINATRA runs with very little performance overhead, adding at most 1.896ms to the execution of event handlers (Section 6.2), which is not noticeable by the user. For realistic user interactions, SINATRA requires less than 6MB of memory to store the events until a future update happens (Section 6.3). Furthermore, the amount of memory grows constantly with the length of active user interactions, with a maximum rate of 253KB/s (Section 6.4), which shows that SINATRA scales well with typical user interactions with modern websites. For websites that make use of frequent XML HTTP Requests (XHR) in the background, SINATRA requires a modest 36MB of storage for a 14h run (Section 6.6). Furthermore, SINATRA supports realistic workloads on modern websites as complex as Twitter, with complex JavaScript that requires over 4500 events to load (Section 6.8).

When performing an update, SINATRA requires at most 1.5 seconds to transfer the state between browsers (Section 6.5.1). We note that the user can continue to interact with the browser during this time. To switch to the updated browser, SINATRA imposes a pause in user interaction of less than 10ms (Section 6.5.2), which is perceived as instantaneous. At its core, SINATRA is an MVX system that delivers events from one browser to another in 19ms or less (Section 6.7).

In short, this paper has the following contributions:

1. The design, and implementation of SINATRA, a system for performing MVX on JavaScript applications.
2. A technique to use SINATRA to perform instantaneous updates to modern commercial closed-source internet browsers, without any loss of state.
3. An extensive evaluation of SINATRA using 4 realistic JavaScript stateful applications and 2 popular websites (Google and Twitter); including widely used JavaScript frameworks Angular [21], JsAction [23], and React [38].

SINATRA's source [1] and research artifact [50] are freely available.

2 Background

Performing *Dynamic Software Updating (DSU)* on a running browser presents many unique challenges. First, state-of-the-art DSU tools require source code changes and do not support programs as complicated as modern internet browsers (Section 2.1). SINATRA circumvents that problem by using *Multi-Version eXecution (MVX)* to perform DSU [44]. Unfortunately, state-of-the-art MVX tools also do not support programs as complicated as modern internet browsers (Section 2.2). SINATRA moves the level of MVX from low-level system calls to high-level JavaScript events. However, performing MVX in the traditional sense is not possible in JavaScript, due to its event-driven paradigm (Section 2.3).

2.1 Dynamic Software Updating (DSU)

Dynamic Software Updating (DSU) allows to install an update on a running program without terminating it, and without losing any program state (e.g., data in memory, open connections, open files). DSU has three fundamental problems to solve: (1) when to stop the running program, (2) how to transform the program state to a representation that is *compatible* with the new version but *equivalent* to the old state, and (3) how to restart the program in the new version. Solving these problems requires modifying the source code of the updatable program [47, 24, 44] adding *safe-update points* to solve problem 1, *state transformation functions* to solve problem 2, and *control-flow migration* to solve problem 3. This is not possible for popular modern internet browsers (e.g., Google Chrome, Microsoft Edge, Apple Safari), as they are closed-source.

Modern DSU approaches focus on *in-process updates* – the new version of the program replaces the old version in the same process – which trivially keep outside resources available between updates (e.g., open network connections and files), but limits existing DSU tools to programs that execute in a single process. This is not the case of modern internet browsers (e.g., Google Chrome uses one process per open tab to improve performance and provide strong isolation between open pages). Finally, modern internet browsers are examples *self-modifying code* given their Just-In-Time (JIT) JavaScript compiler, which is a well known limitation of state-of-the-art DSU tools [24, 44]. Therefore, existing DSU tools cannot update modern browsers.

2.2 Multi-Version eXecution (MVX)

The main goal of MVX is to ensure that many program *versions* execute over the same inputs and generate the same outputs. MVX can be used to perform DSU by launching the updated program as a separate process, transferring the state between processes (e.g., by forking the original process), and resuming execution on the updated process after terminating the outdated process [44].

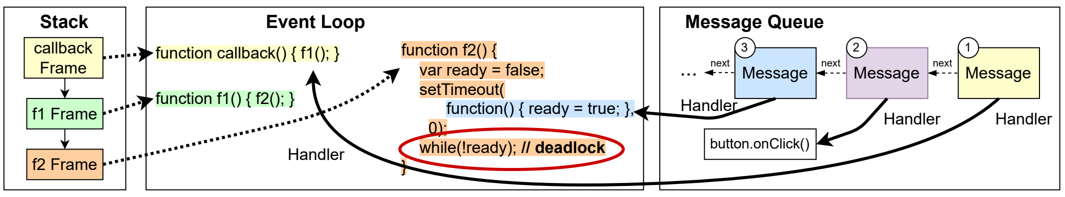
Unlike DSU, state-of-the-art MVX techniques do not require access to the source code of the target program. Instead, MVX interposes *system-calls* through ptrace [35] or binary-code instrumentation [27, 43]. This way, MVX tools can ensure that all processes read the same data, by capturing relevant system-calls (e.g., read) and ensuring that they return the same sequence of bytes.

Unfortunately, existing MVX tools cannot be applied to modern internet browsers. Doing so results in immediate termination due to *benign divergences* – equivalent behavior expressed by different sequences of system calls. For instance, consider how a JIT compiler decides which code to compile/optimize using performance counters based on CPU time. Interacting with such counters does not result in system-calls, and causes JIT compilers to optimize different code, which then results in different system calls. It is possible to tolerate such benign divergences [46], but doing so requires developer support and significant engineering effort, which is not practical.

MVX also suffers from some of the same issues as DSU: no support for multi-process applications, and no support for self-modifying code.

2.3 JavaScript Messages, Event-Loop, and Non-Determinism

JavaScript [16] is an event-driven programming language animated by an *event loop*, as depicted in Figure 1, which processes *messages* from an *event queue*. The event loop takes one message from the event queue and executes its *handler* to completion. If the queue is empty, the event-loop simply waits for the next event. A handler is a JavaScript closure associated with each message. Given that the event loop is *single-threaded*, there is a single *call stack* and one *program counter* (not depicted) to keep the state of processing the current event.



```
function callback() { f1(); }
function f1() { f2(); }
function f2() {
  var ready = false;
  setTimeout(
      function() { ready = true; },
    0);
  while(!ready); // deadlock
}
```

Figure 1 JavaScript's event loop processing 3 messages. Processing Message 1 causes Message 3 to be added to the queue. Message 2 was added when a button was clicked while processing Message 1. Due to JavaScript's event-driven model, this example will never process Message 3, causing a deadlock.

On a browser, events can come from two sources: (1) user interaction with DOM elements (e.g., `onclick` on a `button` element), and (2) browser-generated events, such as expiring timers (e.g., `setTimeout` or `setInterval`) or receiving replies to pending XML HTTP requests. Each event generates a message that keeps track of the event details: the event handler (a closure to be executed for that event), the event target (e.g., the DOM element that generated the message), and other properties. Events handlers in JavaScript are not executed immediately when the event is triggered. Instead, each event is added to the end of the event queue as it is triggered. For instance, in Figure 1, a button was clicked while running function `callback`, which results in adding Message 2 to the event queue.

Events are processed by a single-threaded *event-loop* that runs each event handler to completion before processing any other event, which has two important consequences. **First**, the order and types of events processed are a major part of the non-determinism used to execute a JavaScript program. Apart from asynchronous non-determinism, described below, rerunning the same events in the same order results in the same execution of the same JavaScript program [39, 10, 5, 25, 57]. **Second**, it is not possible for an event handler to issue an event and wait for its completion. This causes the code in Figure 1 to *deadlock* when waiting for the flag `ready` to become `true` [6] because the handler that sets the flag never executes. The handler is associated with a timeout (of zero), which adds a message to the end of the queue. The event loop never finishes executing the current handler, so it never processes any more messages on the queue.

Besides the *synchronous non-determinism* created by events, described above, a JavaScript program can also call functions that are non-deterministic, which we call *asynchronous non-determinism*. The main non-deterministic functions are `Math.random`, which generates random numbers between 0 and 1; and methods of the `Date` object (e.g., `Date.getTime`), which access the current time and date. Notably, it is not possible to seed the pseudo random number generator behind `Math.random`.

Given JavaScript's limitations, **it is not possible to perform traditional MVX on the asynchronous non-determinism.** For instance, when generating a random number, typical MVX approaches ensure each version waits for a message with the same random number (perhaps from a central coordinator process). In JavaScript's case, this would create the same deadlock as shown above. Section 3.4 describes how Sinatra overcomes this limitation.

3 Sinatra Design

Sinatra supports updating internet browsers through a combination of MVX and DSU [44], both at the JavaScript level. Sinatra requires trivial modifications to web pages, which are shown in Figure 2 – Lines 3–4 need to be added. The required changes ensure that

```
01:  <html>
02:      <head>
03: +      <script src="mvx/sinatra.js"       type="text/javascript"></script>
04: +      <script src="mvx/sinatra_init.js"  type="text/javascript" defer></script>
05:      </head>
06:      <body>
07:          <button id="button" onclick="console.log('DOM0 inline')"></button>
08:          <input id="input_textbox" type="text" />
09:
10:          <script>
11:            let button = document.getElementById("button");
12:            button.onmouseover = function(event) { console.log("DOM0"); }
13:
14:            let textbox = document.getElementById('input_textbox');
15:            let closure = function(ev) { console.log('DOM2'); });
16:            textbox.addEventListener('change', closure);
17:          </script>
18:      </body>
19:  </html>
```

Figure 2 Sample HTML code. Sinatra requires adding Lines 3–6.

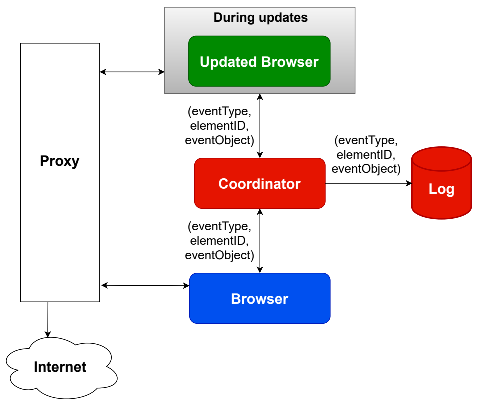

During updates
Updated Browser
Proxy
(eventType, elementID, eventObject)
(eventType, elementID, eventObject)
Coordinator
Log
(eventType, elementID, eventObject)
Browser
Internet

Figure 3 Sinatra architecture.

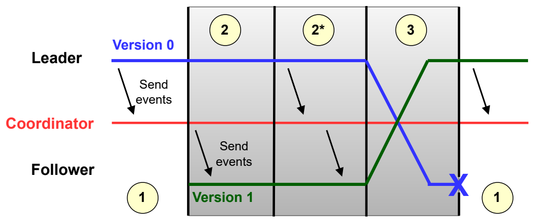


Figure 4 Sinatra update phases. Most of the time is spent in Phase 1. Phase 2 transfers state to an updated browser. Phase 2* is optional, and allows to validate if the update was successful. Phase 3 exposes the updated browser to the user.

Sinatra intercepts event handlers immediately (Line 3) and executes its initialization after the page is loaded but before any other JavaScript code executes (due to the `defer` attribute in Line 4). We note that these are simple modifications that can be performed automatically by a sophisticated proxy [12], as we describe in Section 3.1.

After applying the required changes, Sinatra leverages the first-class nature of functions in JavaScript, and replaces a number of important functions to intercept all sources of non-determinism: `[HTMLElement,HTMDocument].prototype.addEventListener`, `Math.random`, `setTimeout`, `setInterval`, and others; which we describe in Sections 3.3 through 3.5.

3.1 Sinatra Architecture

Sinatra uses three components at all times, shown in Figure 3: (1) the browser, (2) the coordinator, and (3) a proxy. When a browser update is available, Sinatra requires the new version of the browser to be installed at the same time as the old (current) version. The updated browser becomes, temporarily, Sinatra's fourth component. We note that modern browsers can have multiple versions installed side-by-side by performing manual installation into different folders.

Users interact with the browser, which captures JavaScript events and sends them to the coordinator. The coordinator either saves those events in a log, when there is no update taking place, or sends them to the updated browser, thus performing MVX. Users only start interacting with the updated browser after the update is complete, as we describe in Section 3.2.

The proxy serves three main purposes. **First**, the proxy ensures that the updated browser accesses exactly the same resources as the original browser did. Accessing different resources means that each browser processes different JavaScript events and leads to benign divergences, as described in Section 2.2. **Second**, the proxy ensures that outgoing connections remain open while SINATRA changes the roles of the browsers. This is important to ensure that responses to XML HTTP requests are not lost, which can result in errors in the JavaScript application. **Third**, as SINATRA requires minimal changes to the target page, the proxy performs those changes automatically for all the pages accessed by the browser. The proxy must also be able to intercept HTTPS traffic.

There is an off-the-shelf proxy that meets all the requirements: `mitmproxy`[12] can intercept HTTPS traffic (through an extra root certificate), is highly configurable with custom Python code, and can redirect traffic from one connection to another. We validated the feasibility of using `mitmproxy` for SINATRA's purposes through a series of small throw-away prototypes, and we report that `mitmproxy` can indeed be used with SINATRA. However, for the sake of implementation and experimentation simplicity, our current implementation does not use a proxy, as we perform all the changes manually on static HTML pages that do not issue XML HTTP requests.

3.2 DSU with Sinatra

SINATRA performs updates over 3 different phases, as shown in Figure 4.

Phase 1 executes for the vast majority of the time, when no update is taking place. This is the *single-version phase*, which runs a single browser version in isolation. In this phase, SINATRA intercepts all JavaScript events and sends them to a *Coordinator* process, which simply keeps them in memory until the events are needed by later phases. Of course, the coordinator needs to have enough memory to store all the events generated on the browser due to user interaction. Sections 6.3 and 6.4 show that SINATRA's memory requirements are modest, well within the capabilities of modern computers. In instances where a webpage executes network requests in the background, the observed memory requirements remain modest even with a long-running session. A webpage left open will also record any network activity that takes place, such as updates to a live Twitter feed. We describe such an experiment on Section 6.6, which requires 36MB of event storage for a 14h run.

Phase 2 is when updates start, during which the user launches the new browser version. For each page, SINATRA sends all the events from the coordinator and to the new version. Note that SINATRA transfers the state in the background, which allows users to continue to interact with the old browser. Events generated by user interaction during Phase 2 are simply added to the end of the list of events that the new browser needs to process. Section 6.5.1 shows that SINATRA takes, at most, 1.5 seconds in Phase 2.

Phase 2* is optional, and starts when the new browser has processed all the events in the coordinator's log. During Phase 2*, SINATRA performs MVX between the old browser and the new browser. Phase 2* allows to validate whether an update was successful, by comparing each page on the leader with its version on the follower (e.g., matching their DOM tree). If the pages do not match, Phase 2* allows to stop an update that results in loss of data without any disruption, simply by terminating the follower and reverting to Phase 1.

```
1:  let originalHandler = element.onclick;
2:  if (originalHandler) {
3:      closure = function (ev) { id = element.sinatra_id;
4:                                sendToCoordinator("onclick", id, ev);
5:                                originalHandler.call(element, ev); }
6:      registerHandler(element, "onclick", originalHandler);
7:      element.onclick = closure; }
```

Figure 5 Interception of a DOM0 event in the leader. Functions `sendToCoordinator` and `registerHandler` shown in Figure 8 and discussed in Section 3.5.

Validating the pages can be done manually (asking the user if all pages look the same to the user) or automatically (fuzzy matching the DOM contents, or using computer vision algorithms to find differences in screenshots of the rendered pages [25, 67]). Our prototype supports Phase 2* for benchmarking convenience, allowing us to measure the time to transfer logs and to swap roles with great accuracy.

Phase 3 effectively finishes the update by switching the browser exposed to the user. Sinatra *demotes* the old browser version, which becomes the follower, and *promotes* the new browser version, which becomes the leader. At this point, Sinatra can terminate the old browser and start a new *Phase 1*, as the browser was successfully updated with zero downtime and without losing any state. Phase 3 causes the only user-noticeable pause, which we measured in Section 6.5.2 as less than 10ms. For evaluation convenience, our prototype keeps executing the old browser as the follower until the user terminates the old browser.

3.3 Intercepting Events

Sinatra establishes the foundation for MVX and browser updates by intercepting events and sending them from the leader to the follower (through the coordinator). This section explains how Sinatra captures browser events in pure JavaScript by intercepting handlers along with their parameters on the leader.

Sinatra intercepts events by replacing the original event handler with a special handler. This way, when a message causes the event loop to execute a handler, Sinatra's code executes instead, which allows Sinatra to intercept the event that triggered the handler together with the actual handler that is executing. Messages are generated by either DOM0 or DOM2 event listeners:

DOM0 events. DOM0 events can be registered in-line on the HTML page (e.g., Line 7 on Figure 2), and through JavaScript properties on the DOM elements (e.g., Line 12 on Figure 2).

Intercepting DOM0 events is straightforward, as these handlers can be listed/modified directly from DOM elements, simply by reading/writing the respective property, respectively (e.g., Lines 1 and 7 on Figure 5). However, there are two challenges with intercepting DOM0 events. **First,** DOM0 handlers are only present *after* the HTML page loads and executes all in-line scripts. Sinatra's initialization code runs precisely at the right time, just after the HTML page loads but before any handler can be triggered, as shown in Line 3 of Figure 2. At this time, a simple DOM traversal can intercept all inline DOM0 event handlers. **Second**, the page can change DOM0 events without Sinatra noticing. Sinatra uses a mutation listener [60] to register a closure that runs when properties of elements change.

Figure 5 shows how Sinatra uses to intercepts DOM0 events. For each DOM0 handler (Lines 1), Sinatra captures the original handler (Line 2) and replaces it with its own closure (Line 8) that captures the current DOM element – `element` – and the event – `ev`, sends them to the coordinator (Line 5), and runs the handler originally registered by the JavaScript application (Line 6). Note that Sinatra only installs DOM0 events when needed (Line 3).

```
1:  const originalAddEventListener = HTMLElement.prototype.addEventListener;
2:  HTMLElement.prototype.addEventListener = function(evType, evListener, u) {
3:    ownerId = this.sinatra_id;
4:    registerHandler(this, evType, evListener);
5:    let closure = function (ev) { id = ev.target.sinatra_id;
6:                                  sendToCoordinator(evType, ownerId, id, ev);
7:                                  evListener.call(this, ev, u); }
8:    originalAddEventListener.call(this, evType, closure, u); }
```

Figure 6 Interception of DOM2 events on the leader. Functions `sendToCoordinator` and `registerHandler` shown in Figure 8 and discussed in Section 3.5.

DOM2 events. DOM2 events register handlers by calling method `addEventListener` on the target element (e.g., Lines 14–16 on Figure 2). Registering a DOM2 handler requires two arguments: (1) type of event (e.g., `change` for when the target text input box changes), and (2) the event handler itself, specified as a JavaScript closure. Unfortunately, it is not possible to list handlers installed via DOM2. Furthermore, DOM2 describes a complicated logic about how events "bubble" and call all registered event handlers by following the DOM tree and combining DOM0 and DOM2 events. We discuss how bubbling affects Sinatra in Section 4.2. Sinatra intercepts DOM2 events by replacing `[HTMLElement,HTMLDocument].prototype.addEventListener` with its own closure, shown in Figure 6. Note that it is not possible for the underlying JavaScript program to install a DOM2 handler before Sinatra installs its own because Sinatra installs the handler before any other code runs, shown in Line 3 of Figure 2 (scripts without `defer` are downloaded and executed immediately). From this point onwards, when the JavaScript application calls `addEventListener`, Sinatra's code executes instead (Lines 3–8). To intercept DOM2 events, Sinatra installs its own closure using the original `addEventListener` function (Line 8). Then, events that trigger the handler execute Sinatra's closure which starts by sending the event to the coordinator (Line 4) before calling the original handler that the JavaScript application registered (Line 5).

Dynamically created elements. Dynamically created elements can also have event listeners, even before being added to the DOM tree. DOM2 listeners are automatically instrumented, as they use the prototype `HTMLElement` which Sinatra already instruments. Sinatra intercepts DOM0 events through a *Mutation Observer* [60] for new nodes added to the DOM tree, which Sinatra instruments as described above in this section.

Timers. Timers register a closure to execute after a specified time interval through functions `setTimeout` – a one-off event – and `setInterval` – a repeating event, as shown in Lines 3–5 of Figure 7. Of course, such timers are yet another source of synchronous non-determinism that Sinatra must handle. Sinatra uses an approach similar explained above in Section 3.3 and replaces functions `setTimeout` and `setInterval` with Sinatra's own (Line 9). Then, when the underlying application registers a timer, Sinatra transparently intercepts those calls to register its own timer (Lines 13). When the timer expires, Sinatra intercepts the timer event and sends it to the coordinator before executing it (Line 12).

XML HTTP Requests (XHR). XML HTTP Requests (XHR) require an `XMLHttpRequest` object, which defines a number of properties with different roles: (1) hold the data obtained from the remote server (e.g., `status`, `responseText`), or (2) hold closures to be invoked with the XHR changes state (e.g., `onload`, `onreadystatechange`). Sinatra intercepts the function that creates such objects (i.e., `new XMLHttpRequest`) to return a proxy XHR object

```
01:  // Sample program
02:  buttons = [] // A list of buttons
03:  setInterval(30,  function() { // Expensive computation
04:                               r = Math.random(); button[r].enabled = true; });
05:  setInterval(300, function() { buttons[...].enabled = false; });
06:
07:  // Interception
08:  let uniqueID = 0;
09:  let realSetInterval = setInterval;
10:  setInterval = function(origHandler, delay) {
11:      let myID = uniqueID++;
12:      registerInterval(delay, myID, origHandler);
13:      let closure = function() { intervalToCoordinator(myID); origHandler(); };
14:      return realSetInterval(closure, delay); }
```

Figure 7 Sample JavaScript program that uses timeouts (Lines 1–5) and `Math.random` (Line 4); and how Sinatra intercepts timer events (Lines 8–14). Function `registerInterval` and `intervalToCoordinator` are explained Figure 8 and on Section 3.5.

that contains a real XHR object internally. Then, Sinatra defines the same properties as the real XHR object,through `Object.defineProperty`, and uses them to intercept how the JavaScript manipulates the proxy XHR object [14].

As with the DOM events described above, the leader issues XHR requests and sends the returned values to the follower. The follower does not issue any XHR request, it simply gets the data from the leader. When the leader executes a closure associated with XHR state change, it sends information to the follower to trigger the execution of the same closure with the same data. Note that leader and follower issue the same XHR requests by the same order, so Sinatra identifies each XHR request uniquely by the order in which they are issued.

3.4 Intercepting Asynchronous Non-Determinism

As described in Section 2.3, JavaScript programs can call functions that are non-deterministic. The most important such function is `Math.random`, which is used extensively by many JavaScript applications. Unfortunately, using the same approach described in Section 3.3 does not work due to the asynchronous nature of the call to such non-deterministic functions.

For instance, consider the example shown in Figure 7. In this example, there is a list of buttons (Line 2), all disabled. Every 30 seconds, the program performs an expensive computation (Line 3) and enables one button at random (Line 4). Every 5 minutes (300 seconds), the program disables all buttons again (Line 5).

Now consider the following implementation: Sinatra captures the 30 second event, sends it to the coordinator, then captures the execution of `Math.random`, and also sends it to the coordinator. This approach works if all the events are known in-advance (i.e., Phase 2 of Figure 4 and existing record-replay approaches). However, **this approach does not work for MVX** (i.e., Phases 2* and 3 of Figure 4). In this case, it is possible that the follower receives the timer event and reaches the call to `Math.random` before the leader, as the time required to perform the expensive computation may not match in both versions, and the follower may complete it before the leader. At this point, the follower does not know which number to return to match the leader. Making matters even worse, the follower cannot simply wait for the leader, because doing so in JavaScript's event-loop model results in a deadlock, as explained in Section 2.3.

Functions that result in asynchronous non-determinism thus need special consideration. One way to deal with `Math.random` is to use the same seed for the underlying Pseudo-Random Number Generator (PRNG). Unfortunately, it is not possible to seed JavaScript's PRNG.

An alternative is to replace calls to `Math.random` with a custom PRNG that can be seeded, on all variants, which may result in a lower quality source of randomness. Instead, SINATRA starts by generating a sequence of N random numbers when the page loads (i.e., in the `script` tag on Line 3 of Figure 2). Then, each call to `Math.random` consumes one number from the sequence. The leader replenishes the sequence by sending a fresh random number to the coordinator for each number consumed. This approach allows a fast follower to consume up to N random numbers asynchronously at its own pace, ensures all random numbers match between leader and follower, and provides a fresh supply of browser-grade randomness. This approach also simplifies detecting divergences between leader and follower, as we can check in the sequence of random numbers how far/behind one variant is. We found that a cautious value of $N = 100$ works well in practice.

Methods in the `Date` object also require special treatment. The leader starts by consulting the current date/time, saves it in a variable, and sends it to the follower. Then, when the JavaScript program attempts to consult the date, both leader and follower return the saved date, and increment it (e.g., by 100). As such, both versions agree on all dates generated. To ensure fresh and realistic dates/times, every so often, just before sending another message to the follower, the leader refreshes its saved date with the system's and adds date information to the message being sent.

3.5 Multi-Version Execution in JavaScript

So far, this document describes how to capture all the sources of non-determinism used by a JavaScript program on the leader browser. But this is only one half of the problem. To transfer the state between browsers, and to keep them synchronized after that, SINATRA needs to ensure that the follower browser sees exactly the same non-determinism (i.e., the same events in the same order on the same DOM elements).

Matching elements. SINATRA assigns IDs (monotonically increasing numbers) to each DOM element by adding a new property `sinatra_id`, traversing the initial DOM tree after the page is loaded, and then for each dynamically added element (described in Section 3.3). Given that SINATRA traverses the same DOM tree in a deterministic way, and executes `createElement` in the same order in both browsers, the same element always receives the same ID in both browsers.

SINATRA keeps a global structure with all the handlers registered, as shown in Figure 8 (Line 2). When registering events, SINATRA keeps a map for each element ID (Lines 6–9). The map associates event types (e.g., `onclick`) to the respective handler and the target element in which the event was registered (Line 9). We note that each browser keeps references to its own handler and element.

The leader sends events to the coordinator via function `sendToCoordinator`, which serializes the event as discussed in Section 4.2. The follower receives deserialized events from the coordinator via function `receiveFromCoordinator`, which consults the global structure to get the target element (Line 19) and the handler (Line 20) registered for the current event being triggered. Then, the follower calls the handler directly, setting the receiver as the target element (Line 21).

Ensuring the same causal ordering. Given JavaScript's event-driven model, is possible to violate causal ordering in the follower where events are delivered targeting elements that do not exist (yet). For instance, consider a page with one button (1) that, when pressed, creates another button dynamically (2). Consider also an execution in which the user presses buttons

```
01:  // Globals in both leader and follower
02:  let globalHandlerTable = {};
03:  let intervalHandlerTable = {};
04:
05:  // Executed by both leader and follower
06:  function registerHandler(el, eventType, handler) {
07:      let id = el.getAttribute("sinatra_id");
08:      if (!globalHandlerTable[id]) globalHandlerTable[id] = {};
09:      globalHandlerTable[id][eventType] = {"handler": handler, "target": el}; }
10:  function registerInterval(delay, id, handler) {
11:      intervalHandlerTable[id] = { "delay": delay, "handler": handler }; }
12:
13:  // Leader calls:
14:  function sendToCoordinator(eventType, elementID, eventObject) { ... }
15:  function intervalToCoordinator(id) { ... }
16:
17:  // Follower calls:
18:  function receiveFromCoordinator(eventType, elementID, eventObject) {
19:      let targetElement = globalHandlerTable[elementID][eventType]["target"];
20:      let handler       = globalHandlerTable[elementID][eventType]["handler"];
21:      handler.call(targetElement, eventObject); }
22:  function intervalFromCoordinator(id) { intervalHandlerTable[id]["handler"](); }
```

Figure 8 Matching events and timers to handlers and elements in both leader and follower browsers.

(1) and (2). On the follower, it is possible that both button presses are added to the message queue (depicted in Figure 1). Creating the hypothetical button (2) generates another event, which is then added to the message queue *after* the event for pressing that same button (2). This execution violates causal ordering and results in pressing an non-existing button.

To keep causal order consistent between variants, SINATRA uses the latest `sinatra_id` as a logical clock sent with each event from the leader to the follower. Adding new DOM elements results in generating a new `sinatra_id`. When receiving an event, the follower checks the event's `sinatra_id` against the follower's latest `sinatra_id`. If the values do not match, the follower simply postpones processing the event by adding it to the end of the queue with `timeout(0)`, as explained in Figure 1. SINATRA does the same for all events, including XHR requests.

Matching timers. The follower never registers timers and XML HTTP Requests with the browser. Instead, the follower executes the closures registered with each handler in the order that the leader issues them through the coordinator. However, this creates a problem: How can the follower distinguish between many different closures? For instance, consider the example shown in Figure 7. This example installs two closures associated with different timeouts, one in Line 3 and another in Line 5. When one of these expires and the leader executes it, how can the follower know which to execute?

SINATRA uses a unique ID to differentiate each closure registered with a timer (Lines 8 and 11 in Figure 7). Given that SINATRA ensures that the follower executes the same event handlers by the same order as the leader, the IDs always match between variants. Both variants then keep a table from IDs to closures and delays (Line 3 and Lines 10–11 in Figure 8). When sending an event about an expired timer, the leader sends the ID of the closure associated with the timer (Line 12 in Figure 7). The follower then uses the ID to address its table, get the correct closure, and execute it (Line 22 in Figure 8).

Promotion/demotion. When the roles of the browsers switch (Phase 3 in Figure 4), SINATRA uses the information kept on the global structure (Line 2 on Figure 8) to install all event handlers on the respective DOM elements on the promoted browser, and removes them on the demoted browser. SINATRA also cancels all the timers on the demoted browser, and installs them on the promoted browser (with the original timeout value). Because SINATRA does not track how much time passed since each timer was installed, this approach may cause timers to expire after longer than needed (2× in the worst case). However, this is correct as timers in JavaScript guarantee only a minimum amount of time to wait before triggering the associated closure. Pending XHR at the time of promotion cause the leader to postpone the swap until the pending requests are completed. During this time, all new requests are deferred until the follower is fully promoted and eligible to initiate the requests. This leads to user observable pauses until the pending XHR requests are resolved, which Section 6.6 measures as 86ms on a realistic workload.

Read-only Follower. In the context of MVX we now have two browsers, as shown in Figure 4. The user interacts with a *leader* browser, which sends all the non-determinism to the coordinator process. Then, a *follower* browser receives the same non-determinism from the coordinator. This way, SINATRA ensures both leader and follower are always synchronized.

Users can inspect the state of the follower browser, but they cannot modify it because the follower intercepts all the handlers as described in Section 3.3, but does not install any event handlers with the browser. Instead, the follower registers events just with SINATRA (i.e., Line 7 in Figure 5 sets `onclick` to `null`, Line 7 in Figure 6 and Line 13 in Figure 7 are omitted). Also, this approach ensures that the follower executes timer handlers in sync with the leader, running them only when the leader sends the respective events.

4 Implementation

In this section, we describe the implementation details of SINATRA. SINATRA is implemented in pure JavaScript, totaling 2013 lines of code. The web APIs leveraged by SINATRA to intercept user and system generated events are compatible with the most recent versions popular browsers, such as Google Chrome, Mozilla Firefox, Apple Safari, and others. SINATRA works out of the box for most browsers, without requiring external packages, tools, or plugins.

4.1 Coordinator and Protocol

The coordinator process enables communication between both browsers, which is at the core of SINATRA's approach to MVX, and keeps a log of JavaScript events during Phase 1, as shown in Figure 4. We implemented the coordinator process using *node.js* [3], so it executes in its own separate (headless) process without a browser. We use the *SocketIO* [4] JavaScript library to enable bi-directional communication between the coordinator and each browser.

The initialization protocol for SINATRA is quite simple. First, the coordinator should be executing before any browser is launched. On browser launch, SINATRA starts by connecting to the coordinator using a pre-configured address and port, and sends a message. The coordinator replies with the role of this browser, which is *leader* for the first browser and *follower* for the second.

Upon learning its role, a leader browser generates the list of random numbers mentioned in Section 3.4, sends it to the coordinator, and starts sending all events from that point on. A follower browser, conversely, waits for the coordinator to send the list of random numbers, followed by the events that were kept during the leader's execution. At this point, the coordinator informs the follower that it is synchronized, and MVX starts. During MVX, the coordinator sends each event, received from the leader, to the follower as it is received.

Given that communication is bidirectional, the coordinator does not have to establish new channels when promoting the follower/demoting the leader. Instead, each browser simply changes roles and execution continues in MVX, but in the opposite direction. The promotion/demotion event starts from the leader (outdated browser), when the user presses a button that SINATRA injects at the top of the page. Together with a special promotion/demotion message, SINATRA also sends the list of pending timers that were cancelled and their timeouts, as described in Section 3.5.

4.2 Serializing/Deserializing Events and Bubbling

When the underlying JavaScript program executes an event handler on the leader, SINATRA's code is first called with the event. First, SINATRA gets the **name** of the event, (e.g., defined as argument `evType` in Line 2 of Figure 6). Second, SINATRA gets the **ID of the target element** – the element that triggered the change (e.g., defined as argument `element.sinatra_id` in Line 3 of Figure 6). As explained in Section 3.5, SINATRA ensures that all elements have a unique ID. Finally, SINATRA creates a JavaScript object to hold a copy of the event object, and populates it with **all the fields in the event object**, which include the coordinates of mouse events, which key was pressed that triggered the event, and other relevant data.

At this point, SINATRA can send the JavaScript object to the coordinator. The SocketIO implementation automatically turns the JavaScript object into its JSON representation [15] through function `JSON.stringify` on send, and back into a JavaScript object using function `JSON.parse`. The coordinator simply keeps a list of tuples (name, element ID, event) received from the leader. Sending this list to the follower, when it becomes available, requires another round of serializing to JSON by the coordinator, and deserializing back into JavaScript objects by the follower.

An important note is that SINATRA feeds the deserialized event directly into each handler in the follower, as shown in Line 21 of Figure 8. SINATRA does not create/trigger a new synthetic browser event, as some record-replay systems for JavaScript do through `DOMnode.fireEvent` [39]. This design decision simplifies handling *event bubbling*, when many handlers trigger for a single event (e.g., when a child DOM element has a different handler for the same event as its parent). Instead, SINATRA simply captures the order in which the leader executes the event handlers, and their respective targets; and then calls the same handlers by the same order in the follower. The alternative of creating synthetic events has well-known corner-cases that require special consideration. Furthermore, SINATRA can handle browser updates that change the bubbling behavior.

4.3 Stateful DOM Elements and Text Selection

DOM elements, such as radio buttons, check boxes, and text boxes, keep internal state. For instance, when the user selects a check box, the state of that check box changes (it is now selected). Updating the state does not execute any JavaScript handler, which means that SINATRA cannot intercept it directly. Fortunately, there are only a limited number of such elements, and SINATRA handles them as a special case by installing its own event handler associated with the `change` event, even when there is no application handler. The event handler simply captures the updated state of the DOM element, which allows the follower to remain synchronized with the execution on the leader.

Another source of state, and non-determinism, is the text selected by the user. JavaScript can access and manipulate the current text selection, based on a range of characters on a text element (start and end). To detect when the text selection changes, the leader listens for left mouse button releases, and `SHIFT` key releases. At that point, Sinatra can obtain the current text selection (if any), create a JavaScript object that captures the start and span of the selection, and send it to the coordinator. On the follower side, Sinatra uses the information received to select the same text.

5 Practical Considerations and Limitations

The main design goal of Sinatra is to remove all barriers to instantaneous and stateful browser updates, so that users can enjoy automatic browser updates without even noticing them. Another important design goal is to be applicable to all browsers by targeting JavaScript's execution model. This design choice leaves out of scope state maintained inside the browser itself, such as Websockets. For instance, the recent WebRTC standard allows for real-time audio-visual communication, started from JavaScript but implemented inside the browser [63]. Of course, by its design goals, Sinatra does not support such features implemented internally by browsers.

We argue that Sinatra is a practical approach in its current form. Sinatra deals with the browser state that is the most complex and hard to migrate: the JavaScript engine. State-of-the-art DSU approaches excel at dealing with the remaining state inside the browser [24, 36], which would require browser modifications. Perhaps browser vendors can provide an API to migrate open connections and other browser state. Finding such state, and how to migrate it, is exciting future work that is out-of-scope for Sinatra.

Still, Sinatra can deal with pages that hold internal browser state in three possible ways. First, simply reload them. In the WebRTC example, this means that the video-conference connection would drop and reconnect, which is a relatively common event that users tolerate. Second, wait until all such pages are closed and then update the browser. Third, list the unsupported pages and ask users if they accept reloading them.

The current prototype of Sinatra requires two external components: the proxy and the coordinator. We believe that these components can be implemented as plugins to popular browsers [40, 22], and present them here as separate components to highlight the fact that Sinatra works on unmodified browsers.

Other limitations. The main design goal of Sinatra is to allow instantaneous and stateful browser updates. As such, we designed Sinatra under the assumption that only one user interacts with each JavaScript program, and that each JavaScript program does not execute for a long time. All these assumptions break for server-side JavaScript applications written in node.js [3]: many users interact with each JavaScript program, and each program executes for a long time. Even though Sinatra can be applied to such programs, to update the node.js virtual machine, this is not feasible, as such applications handler numerous events within a short time span and result in very large log files. This is outside of the scope of Sinatra.

The current version of Sinatra does not handle persistent state created through `Window.localStorage` [62]. Our evaluation ensures that the persistent state is empty before each run. Supporting persistent state is straightforward: function `Object.keys(localStorate)` can iterate over all the persistent state at the start of execution, and Sinatra can send that to the follower to ensure the same initial persistent state.

Sinatra does not support the Web Workers (WW) API [61], which introduces multi-threading. However, each WW thread executes its own event loop (Figure 1 shows one event loop, each WW has its own event loop), and WWs can only communicate through sending/receiving messages or through a shared `ArrayBuffer` object. Supporting WW requires capturing the total-order of messages sent between different threads, which can be accomplished through Lamport clocks [33, 27, 43, 58]. We plan to add support for WW in future versions of Sinatra.

6 Experimental Evaluation

In this section, we evaluate the feasibility of using Sinatra to deploy browser updates in practice, by measring the overhead it introduces in terms of perceived latency by the user, and extra memory needed by the user's computer. We also evaluate Sinatra's performance as an MVX tool to enable future research. In that regard, we pose the following *research questions (RQs)*:

- **RQ1:** Is the latency added by Sinatra noticeable by the user?
- **RQ2:** What is the average size of the log that Sinatra keeps?
- **RQ3:** How does the size of the log that Sinatra keeps grow with user interaction?
- **RQ4:** How long does Sinatra take to perform a browser update?
- **RQ5:** How much resources does Sinatra require to support XHR on realistic pages?
- **RQ6:** What is the latency when Sinatra is used as a JavaScript MVX system?
- **RQ7:** Can Sinatra be realistically used with modern pages that use complex JavaScript?

We used two versions of two popular internet browsers, Mozilla Firefox versions `82.0` and `83.0`, and Google Chrome versions `88.0.4323.150` and `89.0.4389.72`. Unless when using updates, both leader and follower used the lowest version of each browser. The experimental evaluation took place in a modern desktop computer running Ubuntu Linux 20.04 LTS 64bit, with an Intel(R) Core(TM) i7-9700K CPU 3.60GHz and 32GB of RAM.

6.1 Applications and Workloads

We evaluated Sinatra with the 4 JavaScript applications (describe below). Each application requires user interaction, using the keyboard and/or mouse. We automated such interaction using the tool `Atbswp` [2] to record mouse and keyboard interactions – *workloads* – for each application, and then replay them. `Atbswp` records mouse and keyboard interactions and writes an executable Python script that replays those events using the library `pyautogui` [56]. We now describe each program, and the workload we used:

nicEdit [31] uses JavaScript to add a rich-text editing toolbar to an HTML `div` element. The toolbar applies styles (e.g., bold, italic, underline, font, color, size) to the text selected via the `document.execCommand` JavaScript API [29]. nicEdit creates the toolbar dynamically, using `document.createElement` to generate buttons and custom screens (e.g., to input the URL and text of an hyperlink), and attaches DOM0 event listeners to each generated element (i.e., buttons on the toolbar). nicEdit also creates a `textarea` element dynamically, where the user can input text.

The workload for nicEdit is representative of a user editing text. It starts with a pre-generated text, selects sections of text, and edits each in a different way: making the text bold, italic; changing the font size, font family (Arial, Helvetica, etc), and font format (heading and paragraph). The workload also changes the indentation of a paragraph, increasing it twice and then returning the paragraph back to its original indentation.

DOMTRIS [52] is a dynamic-HTML based Tetris game that uses JavaScript to implement the game mechanics: generate random pieces of different sizes, shapes, and colors; intercept user input; and schedule the movement of each piece inside the Tetris board. DOMTRIS uses `setTimeout` to schedule the downwards movement of the current piece, and `Math.random` to pick the next piece from the available set of pieces. The player interacts with DOMTRIS solely via the directional arrows on the keyboard, intercepted via DOM2 event listeners. Each piece is created dynamically with `document.createElement`.

We automated a Tetris game that drops each piece (without rotating it) on the center, extreme left, and extreme right of the board. As time goes on, the board fills along the center and edges until there is no space left, at which point the game ends. We understand that this is not how people play Tetris, but we cannot use `Atbswp` to automate a valid Tetris game because DOMTRIS uses a different random seed for every game. Even though SINATRA could generate a fixed sequence of random values to ensure deterministic replays, doing so would not show that SINATRA keeps the random values synchronized between browsers.

Painter [49] allows users to draw pictures using the mouse, with various colors and tools (free-hand brushes, lines, rectangles and circles). The user interacts with Painter using only the mouse over 3 HTML5 `canvas` elements [64]: (1) the tool set, (2) the drawing area, and (3) the color and line-width picker. Painter tracks the mouse position and button click/drag using DOM2 events, and reacts to different tool and color selections using DOM0 events. Painter generates a large number of events as it tracks the mouse movements at all times.

Our workload draws a tic-tac-toe board with the brush and line tools, then draws different shapes of different colors inside the board. This requires selecting different tools, colors, and brush strokes; effectively interacting with all parts of Painter. Note that `Atbswp` records the mouse with coarse precision between mouse clicks, which results in a low fidelity replay. For instance, when dragging the mouse along a line, `Atbswp` only captures the mouse position on the start of the line when the mouse button is pressed, one or two positions along the line, and the final position when the mouse button is released. We edited the generated Python script to ensure that the recording replays mouse movements on a pixel-by-pixel basis, to ensure high fidelity and accurate event counts. Unfortunately, due to `pyautogui`'s low performance when replaying a large number of mouse movements, the Painter workload takes minutes to execute what took us seconds to draw.

Color Game [28] is a game that tests reaction time via the Stroop Effect [54] (delay in reaction time between congruent and incongruent stimuli). The game shows players one color name, and requires players to press the button with the same name (out of 4 buttons), but with a different background color (e.g., press the red button with text "Blue" when the game specifies the color "Blue"). The game keeps track of the score (+5 for each correct click, −3 otherwise) during a 30 second round. Color Game is a complex application due to its use of the **Angular JS framework** [**21**]. Internally, Angular uses `document.createElement`, a combination of DOM0 and DOM2 event handlers, `setTimeout`, `setInterval`, and `Math.random`.

The workload consists of one run of the game (30 seconds), clicking each of the four color buttons in arbitrary order after every one second until the game ends.

Google and Twitter are popular pages representative of realistic workloads. The Google search page uses the **JsAction framework** [**23**] and Twitter uses the **React framework** [**38**]. The workload simply allows each page to load and then we interact with each manually.

Table 1 Time required to run event handlers, average of 5 runs with standard deviation.

Program	Browser	Vanilla	Sinatra	Overhead	
				Relative	Absolute
nicEdit	Firefox	$2.600ms \pm 0.144$	$4.496ms \pm 0.371$	$1.729\times$	$+1.896ms$
	Chrome	$3.779ms \pm 0.265$	$4.547ms \pm 0.208$	$1.203\times$	$+0.768ms$
Painter	Firefox	$0.102ms \pm 0.018$	$1.570ms \pm 0.055$	$15.361\times$	$+1.468ms$
	Chrome	$0.114ms \pm 0.004$	$0.982ms \pm 0.080$	$8.639\times$	$+0.869ms$
DOMTRIS	Firefox	$0.495ms \pm 0.101$	$1.497ms \pm 0.094$	$3.025\times$	$+1.002ms$
	Chrome	$0.250ms \pm 0.021$	$0.757ms \pm 0.027$	$3.025\times$	$+0.507ms$
Color Game	Firefox	$1.457ms \pm 0.084$	$1.797ms \pm 0.041$	$1.233\times$	$+0.340ms$
	Chrome	$0.663ms \pm 0.035$	$0.761ms \pm 0.026$	$1.148\times$	$+0.098ms$

6.2 Sinatra Latency

To measure the extra latency added by SINATRA to each event on the leader, we compared the execution of each program without SINATRA (*vanilla*) and with SINATRA. The vanilla version measures the time taken to execute each original event handler during the workload. The SINATRA version measures the time taken to also execute SINATRA's logic together with the original event handler. We measure the runtime of each event handler triggered during the workload, and report the average time among all the event handler executions observed. Table 1 shows the results.

This experiment highlights the extra latency that SINATRA adds to each event. Table 1 shows that SINATRA increases the latency by a maximum of +1.896 (nicEdit on Firefox), from $2.6ms$ to $4.496ms$. The maximum increase in relative terms is for Painter on Firefox, at $15.361\times$, which translates to a low absolute increase of $+1.468ms$, from $0.102ms$ to $1.570ms$. The results answer **RQ1**: **Users cannot notice the extra latency introduced by Sinatra**.

6.3 Log sizes

SINATRA spends the vast majority of the time executing in single-leader mode, as described in Section 3.2. In this mode, SINATRA stores a log in the coordinator with all the events and handlers that the (single) leader executed. In this experiment, we executed the workload for each application in single-leader mode to measure the size of the log on the coordinator, in number of events and size of the log. Table 2 shows the results.

We can see that the number of events varies widely between different experiments. nicEdit has the smallest number of events, as styling text results in a low number of button clicks and text selections. Color Game has twice as many events as DOMTRIS, which involve user input, timers expiring, and random number generation. Finally, as expected, Painter generates the largest number of events due to its fine-grained tracking of mouse events. In terms of absolute log size, we can see that all logs are below 5.4MB. The results of this experiment allow to answer **RQ2**: **Sinatra requires a modest amount of memory to store the log, below 5.4MB per page**. This result shows the practical applicability of SINATRA, given that average modern computers measure memory in tens of GB.

Table 2 Log sizes in single-leader mode, average of 5 runs with standard deviation.

Program	Browser	# of events	Log size (bytes)
nicEdit	Firefox	278 ± 0.0	$142,717 \pm 0$
	Chrome	286 ± 0.0	$122,197 \pm 4$
Painter	Firefox	$2,077 \pm 2.0$	$5,305,731 \pm 5,056$
	Chrome	$2,049 \pm 1.3$	$4,201,712 \pm 2,708$
DOMTRIS	Firefox	683 ± 0.4	$980,094 \pm 46$
	Chrome	682 ± 0.0	$851,230 \pm 9$
Color Game	Firefox	$1,030 \pm 8.7$	$1,533,970 \pm 25,632$
	Chrome	744 ± 4.9	$949,111 \pm 12,141$

6.4 Sinatra scalability

User interactions with websites may differ in length of time and number of events triggered. To measure how SINATRA behaves with different lengths of interaction, we designed an experiment that uses 3 workloads for each application – small, medium, and large – modified as follows.

nicEdit. Repeat the experiment N times, each time performing the same various text changes that have been described previously. **Small:** $N = 2$. **Medium:** $N = 4$. **Large:** $N = 6$.

DOMTRIS. Move Tetris pieces to one, two, or three sides of the board. **Small:** Left side only. **Medium:** Left and right sides. **Large:** Left, right, and center.

Painter. Repeat the drawing N times, pressing the *"Clear"* button (which clears the canvas) in between. **Small:** $N = 2$. **Medium:** $N = 4$. **Large:** $N = 6$.

Color Game. Play a game N times, restarting it at the end of each 30 second run by pressing the *"Restart"* button. **Small:** $N = 1$. **Medium:** $N = 3$. **Large:** $N = 5$.

We repeated the experiment for each size, in single-leader mode, and we measured the duration (seconds), the total number of events in the log (thousands), the size of the log (MB), and the bandwidth needed to send all the events (KB/s). The bandwidth is computed from the duration and the size of the log, and intended to show how much the log grows as the user keeps interacting with a page over time. Table 3 shows the results.

For most experiments, the bandwidth remains roughly constant even as the length of interaction increases, which is to be expected. Color Game is the notable exception, in which the bandwidth increases with the length (and intensity) of user interaction. We believe this is due to internal AngularJS behavior that: (1) never cancels timers with the browser, simply executes a test to return from cancelled timers, which results in more timers expiring as the game is played again and again; and (2) installs `hover` handlers for all elements, which call `Math.random` and result in more handlers executing as the experiment moves the mouse to the *"Replay"* button and back to the playing area. Over time, this results in Color Game generating the largest log files, which is understandable as Color Game is a game that requires intense user interaction. Painter generates large log files because it targets all the mouse movements with a fine level of detail (pixel by pixel, as discussed above). Overall, the bandwidth stays under $253KB/s$, which is acceptable.

We note that the original Painter interaction took about 20sec, the runtimes shown in Table 3 are artificially inflated by the slow speed of `pyautogui`. The original bandwidth would be $1MB/s$, which is acceptable for applications that track the mouse with fine detail.

Table 3 Duration, number of events, log size, and bandwidth needed for growing workloads. **S** means *small*, **M** means *medium*, and **L** means *large*. **FF** means *Firefox*, and **Chr** means *Chrome*. Average of 5 runs.

		nicEdit			DOMTRIS			Painter			Color Game		
		S	M	L	S	M	L	S	M	L	S	M	L
Duration	FF	51	85	119	53	73	91	466	914	1,364	42	122	202
(sec)	Chr	51	85	119	53	73	90	464	921	1,366	43	122	202
# of evts	FF	0.46	0.82	1.18	0.47	0.64	0.63	4.12	8.36	12.14	1.81	9.24	22.53
×1000	Chr	0.48	0.86	1.24	0.47	0.47	0.66	4.09	8.17	12.24	1.76	8.73	21.01
Size	FF	0.26	0.48	0.71	0.67	0.94	1.16	10.56	21.46	31.15	3.05	19.51	50.97
(MB)	Chr	0.22	0.42	0.61	0.58	0.83	1.00	8.40	16.78	25.16	2.59	16.15	41.25
Bandwidth	FF	5.0	5.7	6.0	12.7	12.8	12.8	22.7	23.5	22.8	71.7	159.6	252.3
(KB/s)	Chr	4.3	4.9	5.1	11.0	11.4	11.0	18.1	18.2	18.4	60.9	132.0	204.1

The results of this experiment provide an answer to **RQ3**: **Sinatra logs grow at a rate of 253KB/s as the user interacts with a page.** This result is acceptable, as mouse-based user interactions are short and the bandwidth is not a bottle-neck for inter-process communication. We note that the result is much smaller for all the other cases.

6.5 Browser updates with Sinatra

SINATRA can be used to deploy a browser update without incurring any loss of (JavaScript) state on the pages opened by the running browser. Such updates involve: (1) transferring the JavaScript state to the updated browser, running as follower, by processing the log it receives from the coordinator; and (2) promoting the follower to be the new leader. This section describes two experiments, one for each of the steps.

6.5.1 Log processing time

This experiment measures the time that the updated browser, running as follower, takes to process all the events in the log sent by the coordinator. We executed the workload for each program (to completion), and then launched the new browser as a follower. On the follower, we took two measurements: (1) the time taken since the page is loaded until the follower is up-to-date with the leader, and (2) the time taken just processing the event log sent by the coordinator. Note that (1) includes all the SINATRA initialization logic plus (2). Columns "Process log" and "Start executing" of Table 4 show the results for (2) and (1), respectively.

We can see that processing the log of events is an important portion of the overall time required to start a follower. Most cases take under $338ms$, except Color Game. Color Game takes much longer to process the events in both browsers. We believe this is due to the underlying Angular.JS initializing a large number of libraries it uses as dependencies. Color Game takes longer on Firefox than on Chrome, which we believe is due to Chrome's higher performance when executing Angular.JS code.

6.5.2 Time taken to promote follower

This experiment measures how long it takes to promote the follower to be the new leader (and demote the leader to become a follower) once the follower is up-to-date (i.e., after the follower processes all events sent by the coordinator). The experiment uses two browsers: B_1 as the initial leader, and B_2 as the initial follower. We execute half the workload by

Table 4 Time from launching a follower until its state is up-to-date, time to promote, and round-trip time (RTT) between the leader triggering an event and receiving an acknowledgement from the follower for that event. Average of 5 runs with standard deviation.

Program	Browser	Process log (ms)	Start executing (ms)	Promote (ms)	Round-trip-time (ms)
nicEdit	Firefox	75 ± 6	168 ± 7	8.4 ± 2.3	13.56 ± 0.51
	Chrome	138 ± 7	277 ± 43	8.0 ± 2.0	19.77 ± 2.50
Painter	Firefox	338 ± 56	658 ± 111	6.7 ± 2.4	4.08 ± 0.05
	Chrome	491 ± 23	691 ± 13	4.6 ± 1.6	6.47 ± 1.51
DOMTRIS	Firefox	147 ± 19	420 ± 50	6.6 ± 2.2	32.61 ± 9.10
	Chrome	248 ± 87	438 ± 43	3.8 ± 0.8	26.71 ± 0.09
Color Game	Firefox	$1,067 \pm 103$	$1,435 \pm 129$	7.2 ± 2.9	19.03 ± 1.29
	Chrome	704 ± 52	$1,180 \pm 59$	4.0 ± 2.0	17.15 ± 0.49

Table 5 Latency observed by Sinatra when contacting a server via XHR with a fixed latency, and time require to change roles between variants. Average of 5 runs with standard deviation.

XHR Latency (ms)	Browser	Observed Latency (ms)	Promote (ms)
0	Firefox	7.12 ± 0.34	12.75 ± 1.30
	Chrome	6.60 ± 0.34	11.00 ± 3.08
50	Firefox	56.42 ± 0.42	52.50 ± 2.69
	Chrome	57.27 ± 0.32	56.75 ± 5.07
100	Firefox	107.42 ± 1.13	102.50 ± 1.66
	Chrome	108.84 ± 1.14	111.00 ± 6.78
1000	Firefox	$1,008.25 \pm 0.92$	$1,006.00 \pm 6.20$
	Chrome	$1,009.74 \pm 2.18$	$1,012.25 \pm 9.44$

interacting with B_1, then switch their roles, then finish the workload by interacting with B_2. We checked visually that the experiment behaves as expected, and measure the time taken since switching the roles of each browser. Column "Promote" on Table 4 shows the results. We can see that all promotions happen under $10ms$.

6.5.3 Time to perform an update

Putting together Sections 6.5.1 and 6.5.2 allows us to estimate the minimum time required to perform an update. Even though it may take a follower browser as long as 1.435 seconds to synchronize its state with the leader, this process takes place in the background and does not cause the user to stop interacting with the (leader) browser. Then, once the follower's state is up-to-date, the promote/demote process takes less than $10ms$, which humans perceive as instantaneous. These two experiments also allow us to answer **RQ4**: **Sinatra requires an imperceptible pause (10ms) to update a running browser, and requires less than 1.5 seconds to prepare that update in the background since launching the updated browser.**

6.6 XML HTTP Request support

We evaluate Sinatra's support for XML HTTP Request (XHR) with two experiments. First, we designed an experiment in which a leader and a follower perform 100 XHR requests in sequence to a local server that waits a certain amount of time (0ms, 50ms, 100ms, and 1s) before sending back 100 bytes. On the 50th request, we swap the roles immediately after performing an XHR request, to force Sinatra to postpone the role swap as described in Section 3.3. We measure two things: (1) the latency observed by the leader, and (2) the time required to swap the roles. The results, presented on Table 5, show that Sinatra introduces little extra latency on top of the maximum XHR latency observed. Note that a latency of 100ms is not noticeable by the user.

In our second experiment, we captured all the XHR traffic during a period of 14h on a page that receive very frequent updates – the twitter feeds the local traffic and weather channel[1] – on both Chrome and Firefox. Then, we replay those requests using Sinatra and check the total log size needed. In this experiment, we measured an average number of requests of $5,676$ for (1), $27,022$ for (2); and an average log size under 6MB for (1), and under 36MB for (2). The overall average latency we observed was 86ms. These experiments allow us to answer **RQ5: Sinatra supports realistic XHR with modest storage requirements (under 36MB/14h), and introduces an imperceptible pause due to pending XHR (under 100ms)**.

6.7 Using Sinatra as an MVX system

At its core, Sinatra is an MVX system targeting JavaScript. In this role, we are interested in measuring the latency between an event being triggered on the leader, and that same event being visible on the follower. We designed an experiment that measures the *Round-Trip Time (RTT)* of each event by sending an acknowledgement from the follower, for each event received, back to the leader, through the coordinator. The RTT provides a reasonable estimate of the leader-follower latency. This experiment runs the workloads for all the applications while measuring the RTT. Table 4 shows the results.

In all cases, the RTT is under $33ms$, which indicates a leader-follower latency of half the RTT, around $17ms$. The results from this experiment answer **RQ6**: **Used as an MVX system, Sinatra delivers events to the follower in 10ms after the leader.**

6.8 Using Sinatra on realistic webpages

To test whether Sinatra can be applied to pages that represent a realistic modern workload, we applied it to the Google search page and the Twitter home page (after login, showing a feed of tweets). We downloaded all the resources used by each page in advance, including XHR requests, to be able to observe the same execution reliably. We then modified the downloaded pages to add the required Sinatra headers, as explained in Section 3 and on Figure 2. We used Google Chrome for this experiment, and repeated each experiment 10 times. We measure the time to load each page by adding a closure with `timeout(0)` that readds itself, and measuring the time between each execution. Initially, the time between executions is high as the event-loop is busy loading the page. We measured an idle event-loop imposing a time between executions below $6ms$. When we observe 5 executions below $6ms$, we consider the page loaded.

Loading Google and Twitter, takes $267 \pm 61ms$ and $2891 \pm 321ms$, respectively. Sinatra processes a total of a total of $163{\pm}2$ events in $431{\pm}85ms$ and $4583{\pm}37$ events in $4234{\pm}451ms$, respectively. The overhead is 1.61 for Google, adding about $100ms$; and 1.46 for Twitter, adding about $1sec$.

[1] `https://twitter.com/TotalTrafficCHI`

Once loaded, the pages are fully functional and allow for user interaction. In MVX mode, the interaction on the leader is replicated on the follower without any noticeable delay, confirming that the results in previous experiments generalize to larger pages. Furthermore, we modified our local HTTP server to allow Google XHR traffic to go through, which enabled the search autocomplete feature as the user types to work correctly on the leader, being then replicated on the follower by SINATRA. The results from this experiment answer **RQ7**: **Sinatra can indeed be used on modern pages with sophisticated JavaScript that generate thousands of events with no loss of functionality and modest overhead**.

Threats to validity. Despite our best efforts, the evaluation in this document still has some threats to validity: (1) the websites we tested may not be representative of common websites, (2) the browsers/versions used may not be representative of popular browsers, (3) our results may be affected by bugs in SINATRA, and (4) using SINATRA on other websites may be affected by bugs in SINATRA.

7 Related Work

The problem of Dynamic Software Updating (DSU) has been a focus of past research, resulting in DSU systems for programs written in popular languages such as C [11, 24, 17, 18, 19] and Java [47, 55, 65, 66, 30, 45]. SINATRA differs from these systems in two important ways. First, SINATRA updates the *execution environment* and not the program running on that environment. For instance, DSU systems for Java do not support updating the underlying Java Virtual Machine while running the same program, which would be the closest to the goal of SINATRA. In fact, to the best of our knowledge, SINATRA is the first such DSU system outside of the Smalltalk community [20, 9] to target the execution environment specifically. Second, DSU systems typically require modifications to the programs being updated to support stopping the program in one version and resuming it in the next, and to express how to transform the state in the old program to an equivalent representation that is compatible with the updated code. In contrast, SINATRA works on unmodified closed-source commercial internet browsers. Instead of migrating the state directly, SINATRA launches the new browser as a separate process and migrates the state for each page individually. The only state kept outside of SINATRA is persistent HTTP connections, which SINATRA's proxy keeps open during updates.

SINATRA uses Multi-Version eXecution (MVX) to synchronize the old and new versions of the updated browser. MVX has been used mostly in programs written in C/C++ by intercepting and synchronizing system-calls between processes. The main goal of MVX are: (1) to increase security [32, 13, 59], detecting divergences in potentially suspect processes; (2) to increase reliability [27, 8, 34, 37, 51], tolerating faults in one process by using the other processes; and (3) availability [26, 44, 48], by performing updates on a forked process and terminating it when updates fail, without any disruption. In fact, Mvedsua [44] is the most similar MVX system to SINATRA, given that it also combines MVX with DSU and allows users to build confidence on the validity of the update by executing both old and new versions for a period of time. However, Mvedsua targets C programs updated via Kitsune [24], intercepts system calls, requires modifications to the programs being updated and machine-parseable descriptions of the update-induced divergences. SINATRA requires much less developer effort, which can be fully automated using an HTTP proxy.

Record-Replay (RR) can be described as "offline Multi-Version eXecution". It operates in two phases, typically using two different (automatically generated) programs. First, it records all non-determinism observed during an execution in a log file. Then, it uses that log file to replay the same execution over the same program. By contrast, MVX records each non-deterministic datum in one process and replays it immediately on another process, thus keeping the state on both processes perfectly synchronized. MVX also needs to account for differences in execution speed that may result in a replayer overtaking the recorder and reaching a program point that requires non-determinism before that data is available. For this reason, RR approaches require the log to be complete before being able to replay it. Furthermore, RR approaches do not allow a replayer to become a recorder as they target a different problem: Accurate replication of bugs observed in production during development.

Techniques for RR target programs written in multiple popular programming languages: C [42], Java [7], and even web browsers [39, 10, 5, 25, 57]. Most techniques require a modified web browser. Dolos [10] and Jardis [5] use modified implementations of browser components (Webkit and ChakraCore) to record bugs in production and replay them in development and provide developers with time-travel debugging capabilities, respectively. Jardis focuses on node.js applications [3]. ReJS [57] also provides support for time-travelling debugging, but for any JavaScript code in general, through a modified version of Microsoft's ChakraCore JavaScript engine that performs heap checkpoints via a modified garbage-collector.

Working in pure JavaScript, Mugshot [39] is an RR system that demonstrates the feasibility of capturing all the needed non-determinism to ensure an accurate replay without the need for a modified web browser. Mugshot influenced the design of Sinatra by listing all the sources of non-determinism that need to be handled to capture all interactions between the environment and a JavaScript program executing in a browser. However, Mugshot relies on event listeners on the topmost DOM element (i.e., `window`) to intercept all events, and replays them through synthetic browser events (i.e., `DOMElement.fireEvent`). As a result, Mugshot has to deal with browser-specific behavior that impacts event bubbling and event ordering. Sinatra's approach of intercepting each handler individually avoids such complexity and naturally supports any browser without special handling. Similarly to Sinatra, X-Check [25] also works in pure JavaScript and works on different browsers (all other techniques require the same browser and version to replay the recorded logs). X-Check records logs on one browser and replays them on different browsers, with the goal of detecting cross-browser differences that developers can then replicate and address.

The closest system to Sinatra is Cocktail [67], an MVX system for web browsers with the goal of improving the security of internet browsers by feeding input to many different browsers and voting on the output. Cocktail can thus detect and defeat attacks that target a particular browser, or a particular browser version. Despite the very different goal, there are more important differences between Sinatra and Cocktail. First, Cocktail is implemented as a browser plugin and Sinatra is implemented in pure JavaScript. As such, Sinatra can be directly applied to any web browser, but Cocktail requires developer effort to write a new plugin for that browser. Also, Cocktail's plugin can intercept asynchronous non-determinism, such as calls to `Math.random`, and block until all browsers reach the same point. This is not possible in JavaScript's execution model, as described in Section 2.3. Second, Cocktail relies on an UI component to intercept mouse and keyboard events before they reach the browser. Sinatra captures the events at a finer level of detail, ensuring that all browsers execute the same JavaScript handlers by the same order, regardless of implementation-specific browser quirks that may show the same element on different positions in different websites. In fact, Sinatra can replicate the execution even if the leader and follower have different window dimensions, which is a limitation of Cocktail.

8 Conclusions and Future Work

This paper presented the design of Sinatra, a system that allows to update internet browsers without losing any state in the process. Sinatra works fully at the JavaScript level, using first-class function interception to keep track of all events, and then using MVX to perform updates on the new version of the browser while the old version keeps providing service. As a result, Sinatra works on popular, closed-source, commercial internet browsers such as Google Chrome. Sinatra requires a small amount of JavaScript source changes, performed to each page opened in the target browser. The changes required are easy to automate with a sophisticated internet proxy.

This paper also presented an extensive experimental evaluation, where Sinatra is applied to JavaScript applications with different combinations of features. When not performing an update, Sinatra imposes low overhead on the execution of event handlers (a max increase of 2.107ms). Also, the state that Sinatra keeps to support future updates grows at a modest rate of 10.8KB/s (at most) during intense user interaction. If a page remains open performing XML HTTP Request requests, Sinatra requires a modest 36MB of storage for a 14h run.

Sinatra can perform updates in short order, requiring just 1.5s (at most) to transfer the state from the old browser to the new browser. While Sinatra transfer the state, the user keeps interacting with the old browser. Then, to finish the update and allow the user to interact with the new browser version, Sinatra requires a very short pause in user interaction of less than 10ms, which is barely noticeable for most users.

Besides its role in browser updates, Sinatra doubles as an MVX tool for JavaScript applications. The experimental evaluation showed that Sinatra can keep two browsers synchronized, with an action on one browser taking effect on the other almost instantaneously, within 19ms.

In the modern internet age, an up-to-date internet browser is the first line of user defense. Sinatra dramatically lowers the barrier to deploy automatic and fully transparent browser updates by eliminating any data loss or service interruption associated with such updates. We strongly believe that Sinatra has the potential to improve the average user's safety by making disruptive browser updates a thing of the past.

Future Work. It is possible to use Sinatra to move a page from one browser to another (e.g., from Mozilla Firefox to Google Chrome). This feature can be valuable to security conscious users, who can switch browsers as a vulnerability is disclosed. We tested this feature of Sinatra informally to ensure it works, but did not evaluate it or develop it further.

Sinatra is OS and platform agnostic, and we plan to implement Sinatra on popular platforms (e.g., Microsoft Windows and Apple OSX) and apply Sinatra to the official browser in each platform (e.g., Microsoft Edge and Apple Safari).

References

1 Sinatra's github reposiroty. `https://github.com/bitslab/sinatra`.

2 Automate the boring stuff with python. `https://github.com/RMPR/atbswp`, 2021. Accessed: 2021-04-14.

3 Node.js. `https://nodejs.org/en/`, 2021. Accessed: 2021-04-14.

4 Socket.io. `https://socket.io/`, 2021. Accessed: 2021-04-14.

5 Earl T. Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. Time-travel debugging for javascript/node.js. In *FSE '16 Proceedings of the 2016 ACM International Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, September 2016. URL: `https://www.microsoft.com/en-us/research/publication/time-travel-debugging-javascriptnode-js/`.

6 Samuel Baxter, Rachit Nigam, Joe Gibbs Politz, Shriram Krishnamurthi, and Arjun Guha. Putting in all the stops: Execution control for javascript. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 30–45, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3192366.3192370.

7 Jonathan Bell, Nikhil Sarda, and Gail Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 362–371. IEEE Press, 2013.

8 Emery Berger and Benjamin Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, volume 41, pages 158–168, January 2006. doi:10.1145/1133255.1134000.

9 Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example.* Square Bracket Associates, 2009. URL: http://pharobyexample.org.

10 Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. Interactive record/replay for web application debugging. In *UIST 2013: Proceedings of the 26th ACM Symposium on User Interface Software and Technology*, pages 473–484, St. Andrews, UK, October 2013.

11 Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. Polus: A powerful live updating system. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 271–281, USA, 2007. IEEE Computer Society. doi:10.1109/ICSE.2007.65.

12 Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. mitmproxy: A free and open source interactive HTTPS proxy, 2010–. [Version 5.3]. URL: https://mitmproxy.org/.

13 Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, USA, 2006. USENIX Association.

14 ECMA (European Association for Standardizing Information and Communication Systems). Standard ECMA-262 6th Edition – Section 19.1.2.4. https://262.ecma-international.org/6.0/#sec-object.defineproperty. Accessed: 2022-01-04.

15 ECMA International. Standard ECMA-404 – The JSON data interchange syntax. https://www.ecma-international.org/publications-and-standards/standards/ecma-404/, December 2017.

16 ECMA International. Standard ECMA-262 – ECMAScript(R) 2020 language specification. https://www.ecma-international.org/publications-and-standards/standards/ecma-262/, June 2020.

17 Cristiano Giuffrida, Călin Iorgulescu, Anton Kuijsten, and Andrew S. Tanenbaum. Back to the future: Fault-tolerant live update with time-traveling state transfer. In *Proceedings of the 27th USENIX Conference on Large Installation System Administration*, LISA'13, pages 89–104, USA, 2013. USENIX Association.

18 Cristiano Giuffrida, Călin Iorgulescu, and Andrew S. Tanenbaum. Mutable checkpoint-restart: Automating live update for generic server programs. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 133–144, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2663165.2663328.

19 Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and automatic live update for operating systems. *SIGARCH Comput. Archit. News*, 41(1):279–292, March 2013. doi:10.1145/2490301.2451147.

20 Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

21 Google. Angularjs. https://angularjs.org/, 2018. Accessed: 2021-04-14.

22 Google Inc. API Reference – Chrome Developers. `https://developer.chrome.com/docs/extensions/reference/`. Accessed: 2022-01-04.

23 Google Inc. JsAction repository. `https://github.com/google/jsaction`. Accessed: 2022-01-04.

24 Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for C. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, October 2012.

25 M. He, G. Wu, H. Tang, W. Chen, J. Wei, H. Zhong, and T. Huang. X-check: A novel cross-browser testing service based on record/replay. In *2016 IEEE International Conference on Web Services (ICWS)*, pages 123–130, 2016.

26 Petr Hosek and Cristian Cadar. Safe software updates via multi-version execution. In *International Conference on Software Engineering (ICSE 2013)*, pages 612–621, May 2013.

27 Petr Hosek and Cristian Cadar. Varan the unbelievable: An efficient n-version execution framework. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015)*, pages 339–353, March 2015.

28 Linghua Jin. Stroop effect color game build with angularjs. `https://github.com/linghuaj/Angular-ColorGame`, 2016. Accessed: 2021-04-14.

29 Aryeh Gregor Johannes Wilm. execcommand – unofficial draft 13 april 2021. `https://w3c.github.io/editing/docs/execCommand/`, 2021. Accessed: 2021-04-14.

30 Jevgeni Kabanov and Varmo Vene. A thousand years of productivity: the jrebel story. *Software: Practice and Experience*, 44(1):105–127, 2014. `doi:10.1002/spe.2158`.

31 Brian Kirchoff. Nicedit – wysiwyg content editor, inline rich text application. `https://nicedit.com/`, 2008. Accessed: 2021-04-14.

32 Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *Proceedings - 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016*, pages 431–442. Institute of Electrical and Electronics Engineers, Inc., September 2016. `doi:10.1109/DSN.2016.46`.

33 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. `doi:10.1145/359545.359563`.

34 Liming Chen and A. Avizienis. N-version programminc: A fault-tolerance approach to rellablllty of software operatlon. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'.*, pages 113–, 1995. `doi:10.1109/FTCSH.1995.532621`.

35 Linux Foundation. ptrace – linux standard base core specification 4.1. `http://refspecs.linux-foundation.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic/baselib-ptrace-1.html`, 2010. Accessed: 2021-04-14.

36 Kristis Makris and Rida A. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, page 31, USA, 2009. USENIX Association.

37 Matthew Maurer and David Brumley. Tachyon: Tandem execution for efficient live patch testing. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 617–630, Bellevue, WA, August 2012. USENIX Association. URL: `https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/maurer`.

38 Meta Platforms, Inc. React – A JavaScript Library for building user interfaces. `https://reactjs.org/`. Accessed: 2022-01-04.

39 James Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for javascript applications. In *Proceedings of NSDI*. USENIX, April 2010. URL: `https://www.microsoft.com/en-us/research/publication/mugshot-deterministic-capture-and-replay-for-javascript-applications/`.

40 Mozilla Inc. Browser Extensions – Mozilla MDN. `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions`. Accessed: 2022-01-04.

41 Mozilla Inc. Firefox Public Data Report. `https://data.firefox.com/dashboard/user-activity`. Accessed: 2022-01-04.

42 Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Lightweight user-space record and replay. *CoRR*, abs/1610.02144, 2016. `arXiv:1610.02144`.

43 Luís Pina, Anastasios Andronidis, and Cristian Cadar. Freeda: Deploying incompatible stock dynamic analyses in production via multi-version execution. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '18. ACM, May 2018.

44 Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. MVEDSUa: Higher Availability Dynamic Software Updates via Multi-Version Execution. In *Proceedings of the ACM 24th Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19. ACM, April 2019.

45 Luís Pina and João Cachopo. Atomic dynamic upgrades using software transactional memory. In *Proceedings of the 4th International Workshop on Hot Topics in Software Upgrades*, Hot-SWUp. IEEE, June 2012.

46 Luís Pina, Daniel Grumberg, Anastasios Andronidis, and Cristian Cadar. A dsl approach to reconcile equivalent divergent program executions. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC '17. USENIX, July 2017.

47 Luís Pina, Luís Veiga, and Michael Hicks. Rubah: DSU for java on a stock JVM. In *Proceedings of the ACM 2014 International Conference on Object-Oriented Programming Languages, Systems, and Applications*, OOPSLA '14. ACM, October 2014.

48 Weizhong Qiang, Feng Chen, Laurence T. Yang, and Hai Jin. Muc: Updating cloud applications dynamically via multi-version execution. *Future Generation Computer Systems*, 74:254–264, 2017. `doi:10.1016/j.future.2015.12.003`.

49 Rafael Robayna. Canvas painter. `http://caimansys.com/painter/`, 2006. Accessed: 2021-04-14.

50 Ugnius Rumsevicius, Siddhanth Venkateshwaran, Ellen Kidane, and Luís Pina. Artifact for SINATRA: Stateful Instantaneous Updates for Commercial Browsers through Multi- Version eXecution, February 2023. `doi:10.5281/zenodo.7647070`.

51 Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In Wolfgang Schröder-Preikschat, John Wilkes, and Rebecca Isaacs, editors, *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*, pages 33–46. ACM, 2009. `doi:10.1145/1519065.1519071`.

52 Jacob Seidelin. DOMTRIS – A DHTML Tetris clone. `https://web.archive.org/web/20140805202021/http://www.nihilogic.dk/labs/tetris/`, 2014. Accessed: 2021-04-14.

53 StatCounter GlobalStats. Desktop Browser Version Market Share Worlwide. `https://gs.statcounter.com/browser-version-market-share/desktop/worldwide//#daily-20201001-20201201`. Accessed: 2022-01-04.

54 J. R. Stroop. Studies of interference in serial verbal reactions. *Journal of Experimental Psychology*, 1935. `doi:10.1037/h0054651`.

55 Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: A VM-centric approach. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, June 2009.

56 Al Sweigart. Pyautigui documentation. `https://pyautogui.readthedocs.io/en/latest/`, 2019. Accessed: 2021-04-14.

57 John Vilk, James Mickens, and Mark Marron. A gray box approach for high-fidelity, high-speed time-travel debugging. Technical Report MSR-TR-2016-7, Microsoft, June 2016. URL: `https://www.microsoft.com/en-us/research/publication/gray-box-approach-high-fidelity-high-speed-time-travel-debugging/`.

58 Stijn Volckaert, Bart Coppens, Bjorn De Sutter, Koen De Bosschere, Per Larsen, and Michael Franz. Taming parallelism in a multi-variant execution environment. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 270–285, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3064176.3064178.

59 Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. Secure and efficient application monitoring and replication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 167–179, Denver, CO, June 2016. USENIX Association. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/volckaert.

60 W3C. DOM – Living Standard – Section 4.3: Mutation Observers. https://dom.spec.whatwg.org/#mutation-observers. Accessed: 2022-01-04.

61 W3C. HTML – Living Standard – Section 10: Web workers. https://html.spec.whatwg.org/multipage/workers.html#workers. Accessed: 2022-01-04.

62 W3C. HTML – Living Standard – Section 12: Web storage. https://html.spec.whatwg.org/multipage/webstorage.html#webstorage. Accessed: 2022-01-04.

63 W3C. WebRTC 1.0: Real-Time Communication Between Broswers. https://w3c.github.io/webrtc-pc/. Accessed: 2022-01-04.

64 Web Hypertext Application Technology Working Group (WHATWG). Html living standard – 4.12.5 the canvas element. https://html.spec.whatwg.org/multipage/canvas.html#the-canvas-element, 2021. Accessed: 2021-04-14.

65 Thomas Würthinger, Danilo Ansaloni, Walter Binder, Christian Wimmer, and Hanspeter Mössenböck. Safe and atomic run-time code evolution for java and its application to dynamic aop. *SIGPLAN Not.*, 46(10):825–844, October 2011. doi:10.1145/2076021.2048129.

66 Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic code evolution for java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 10–19, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1852761.1852764.

67 Hui Xue, Nathan Dautenhahn, and Samuel T. King. Using replicated execution for a more secure and reliable web browser. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012. URL: https://www.ndss-symposium.org/ndss2012/using-replicated-execution-more-secure-and-reliable-web-browser.