A Gray Box Approach For High-Fidelity, High-Speed Time-Travel Debugging

John Vilk
University of Massachusetts Amherst
jvilk@cs.umass.edu

James Mickens
Harvard University
mickens@g.harvard.edu

Mark Marron
Microsoft Research
marron@microsoft.com

Abstract

Time-travel debugging (TTD) lets developers step backward as well as forward through a program's execution. TTD is a powerful mechanism for diagnosing bugs, but previous approaches suffer from poor performance due to checkpoint and logging overhead, or poor fidelity because important information like GUI state is not tracked.

In this paper, we describe how to provide high-performance *and* high-fidelity TTD to programs written in managed languages. Previous high-performance debuggers treat components external to the program like the GUI as black boxes, but that is not sufficient for high-fidelity time-travel. Instead, we advocate for a gray-box approach that keeps these components live and in sync with the program during time-travel. The key insight is that managed runtime APIs expose *most* of the functionality required to do this; where it does not, we extend the runtime with a small number of non-intrusive *interrogative interfaces*. To demonstrate the power of our gray-box approach, we implement REJS, a time-traveling debugger for web applications. REJS imposes imperceptible tracing overhead, and its logs typically grow less than 1 KB/s. As a result, REJS is performant enough to be deployed in the wild; real client machines can ship buggy execution traces across the wide area to developer-side machines for debugging.

1 Introduction

Developers spend a large amount of time debugging. To fix a particular bug, a developer first determines the inputs that trigger the problem. Then, the developer launches the program within a debugger and find the root cause of the bug. If the developer steps too far forward in the program, or fails to place a breakpoint in the correct location, the developer may pass by the problematic line of code; the developer will then have to adjust her breakpoints and restart the debugging process.

Time-traveling debuggers [23, 14, 5] offer the promise of removing much of this frustration. Using these systems, a developer only needs to record a problematic execution once; the developer can then step forward *and backward* through the recorded execution. If time travel were sufficiently fast, it could be used as a primitive by question-guided debuggers [24, 25] and automated root cause extractors [9, 16, 20, 22]. If execution logs were sufficiently small, they could be shipped across the wide area, letting developers receive bug traces from real end-users [32]. If replay fidelity were sufficiently high, developers could use time-travel debugging to diagnose race conditions and other subtle faults.

Unfortunately, there is a tension between the fidelity of replay and the efficiency of program tracing and subsequent time travel. For example, a hypervisor has an omniscient view of how a guest application interacts with the rest of the system. Thus, a hypervisor can record all of the nondeterministic inputs to the guest, and later perform instruction-precise replay [14, 23]. The replay will accurately reconstruct low-level hardware and OS state, but knowledge of such state is often unnecessary for an application-level developer who is trying to fix an application-level bug. In these debugging scenarios, the performance and space overheads of instruction-level TTD are excessive.

To reduce these overheads, one can define a higher-level virtual machine boundary. For example, a time-traveling debugger can mediate a program's interactions with the POSIX interface [43]; alternatively, the debugger can mediate interactions with a managed language runtime like .NET or JVM [5]. With fewer, higher-level interactions to log, execution tracing is more efficient, and logs are smaller. Unfortunately, a high-level virtual machine interface excludes important application state that may be needed at debug time. For example, consider a debugger that logs and replays interactions with the POSIX layer. Using calls like `fcntl(int fd,...)`, a program can query the state associated with a file. At logging time,

the debugger can record the return values for such calls, but fundamentally, the debugger must treat the file system as a *black box*—the underlying file system state is maintained outside of the virtual machine boundary, within the OS. Thus, the debugger has no insight into how the file system's state might change between explicit application queries of that state. If the program fails to acquire a lock on a file, the developer may want to call `fcntl` during replay to query for file locks set by other processes, posing a challenge to the debugger. The debugger can disallow the query by fiat (and thereby diminish the quality of the debugging experience). Alternatively, the debugger can maintain an internal model of how the OS might update the file system, and return an `fcntl()` value from the simulated replay-time file system. Unfortunately, simulating program-external components is brittle and difficult to implement correctly [8, 32].

Given the preceding discussion, there seems to be a stark partition in the design space for time-traveling debuggers: precise but heavyweight, or imprecise but efficient. In this paper, we demonstrate that this partition is not fundamental. We introduce a new type of time-travel debugger which has the fidelity associated with low-level logging, but the efficiency associated with high-level logging. The key insight is that black-box components below the virtual machine boundary can be modified to expose *just enough additional state* to let high-level logs support high-fidelity replay; we call this approach *gray-box virtualization*.[1] The amount of additional exposed state is small; thus, execution tracing remains efficient, and logs remain small. For example, we could augment the POSIX interface to fire upcalls whenever a file's lock set changes, letting the logging infrastructure track modifications to formerly hidden file state.

That being said, this paper focuses on gray-box virtualization at the managed runtime layer, not the POSIX layer. There are two reasons for this approach. First, the necessary gray-box modifications are simpler to enumerate and easier to correctly implement at the higher level of abstraction that managed runtimes provide. Second, by logging and replaying at this higher level of abstraction, we demonstrate that high-fidelity time-travel debugging becomes so efficient that it is deployable in production environments at production speeds: real end-users can enable logging with imperceptible performance impact; logs are small enough to send to remote developers over poor network connections; and developers can replay those buggy executions in real-time or faster. We demonstrate the effectiveness of gray-box virtualization with REJS, a new time-traveling debugger for client-side web applications that uses the JavaScript runtime as the virtualization layer.

[1] This modification of black-box components is similar in spirit to paravirtualization [4], which modifies a guest OS to be more amenable to x86-level virtualization.

REJS leverages gray-box techniques to capture important state like animation metadata that resides in the (formerly black box) rendering engine. REJS can faithfully replay interactions that previous high-performance JavaScript debuggers cannot (§3.2) while avoiding the heavyweight tracing of previous high-fidelity approaches (§2), producing application snapshots that are *three orders of magnitude* smaller than snapshots from Hyper-V, a state-of-the-art, x86-level hypervisor. Furthermore, our unoptimized REJS prototype can generate a full application checkpoint in a few hundred *milliseconds*. For many use cases, such fast checkpoints obviate the need for the complex snapshotting techniques that are required in x86-level virtualization environments [10, 45].

In summary, this paper provides four contributions.

- We introduce gray-box virtualization and describe how it enables high fidelity TTD without the performance overheads of low-level execution tracing (§3).
- We describe REJS, a concrete implementation of a gray-box virtualization system (§4), and evaluate its efficiency and fidelity on a wide variety of applications (§5).
- We demonstrate that gray-box virtualization enables other services besides TTD, including cheap application migration (§4.5).
- We discuss how gray-box virtualization could be adapted to other managed runtimes, such as the .NET Common Language Runtime and the Java Virtual Machine (§6).

REJS has been incorporated as part of the debugging platform of the open-source ChakraCore JavaScript engine from Microsoft [36].

2 Background

In the context of time-travel debugging, a program has two types of state. *Internal state* resides above the virtualization boundary, and can be explicitly manipulated and inspected by a program. For example, in a POSIX application, the user-mode memory pages and the open file descriptors are internal state. Managed languages do not expose raw memory or file descriptors, but the analogues (e.g., object references and IO objects) belong to the internal state of the managed program.

External state resides beneath the virtualization boundary, and is partially or totally hidden from the program, or updated by events that are not directly exposed to the program. In the POSIX example from Section 1, a file descriptor's lock metadata is external state: a process may query or update lock state, but that state can be concurrently modified by a second process without triggering an explicit event in the first one. In a managed language, GUI resources are a classic example of external state.

		Application-facing Interface		
Resource	**State**	**Web Browser**	**JVM**	**.NET CLR**
CPU	Thread Status	None	`ThreadGroup`	`Process.Threads`
	Active IPC Listeners	`postMessage` listeners	None	`IpcChannel`
	RNG State	`Math.random()`	`Math.random()`	`Random`
	Thread Locks	None	`Object`	`Monitor`
	Perf. Monitoring	None	`sun.management.counter.*`	`PerformanceCounter`
Memory	Heap	Indirect	Indirect	Indirect
	Stack	Indirect	Indirect	Indirect
Storage	Storage Contents	`localStorage`, Cookies	`FileSystem`	`System.IO.*`
Clock	Current Time	`Date`	`System.currentTimeMillis`	`DateTime`
	Timer Status	`setTimeout`, `setInterval`	`Timer`	`Timer`
Display	GUI Contents	DOM	AWT, Swing	WPF, WinForms, WinRT
Device Input	Event Listeners	DOM Events	`EventListener`	`KeyEventHandler`, ...
	Pending Events	Indirect	`EventQueue`	`Dispatcher`
Network	Connection Status	`XMLHttpRequest`	`Socket`	`Socket`
	Data Listeners	DOM Events	`AsynchronousSocketChannel`	`Socket` + `AsyncCallback`

Table 1: Managed runtimes provide a high-level interface to low-level system resources. In the table above, we summarize those resources, and provide examples of the APIs which expose those resources. In the "interface" columns, *Indirect* means that a program's interactions with a low-level resource are implicit, i.e., the interactions do not use an explicit interface in the managed runtime. *None* indicates that the runtime does not expose a particular resource at all.

GUIs are typically implemented in native code, by concurrently executing processes that live outside of the managed runtime's interpreter/JIT environment. As a result, a managed program lacks access to low-level GUI events, rendering buffers, or animation timers, *even though the program can indirectly read and write that state.* Thus, logging that state is crucial for accurately replaying the program (§3.2).

By interposing on the virtualization boundary, a TTD framework can trace and replay interactions across the boundary. The primary challenge of efficient TTD is identifying the minimal amount of internal plus external state which must be tracked to faithfully replay a program. TTD performance suffers if a framework logs and replays *extraneous* state, i.e., state which is unnecessary for understanding the program under investigation.

In the rest of this section, we provide a more detailed overview of how managed runtimes expose low-level resources. We then describe how prior solutions for time-travel debugging deal with the problem of external state; this discussion motivates gray-box virtualization, which is described in depth in Section 3.

2.1 Managed Runtimes

Table 1 compares the interfaces for three managed runtimes: the JavaScript runtime that is exported by web browsers [15], the Java Virtual Machine (JVM) [26], and the .NET Common Language Runtime (CLR) [33]. Each runtime uses idiosyncratic methods to expose the same low-level OS resources, but most resources are exposed in equivalently expressive ways across each runtime. However, the JavaScript interface is higher-level than that of the JVM or CLR. As a result, JavaScript programs have more external state than semantically equivalent JVM/-CLR programs. Furthermore, a JavaScript program cannot express some behaviors that are possible in the other runtimes. For example, the JavaScript runtime does not provide direct access to the host file system. Instead, a JavaScript program must interact with persistent storage via key/value interfaces like `localStorage`; those interfaces are private, per-origin resources, meaning that two origins that need to communicate must use IPC as there is no shared file system. The JVM and CLR runtimes also allow a single execution context to contain multiple threads, whereas JavaScript does not. Thus, the JVM and CLR runtimes expose locking primitives that are unnecessary for JavaScript programs.

All of the managed runtimes provide an abstracted view of memory. Raw memory addresses are hidden, and programs use opaque object references to manipulate data. Such memory abstraction has an important ramification for time-travel debugging: at logging time, the debugger does not need to track address-precise object

locations, and at replay time, the debugger does not need to recreate objects in their exact logging-time memory locations. Instead, replay only needs to provide *application-visible referential integrity*. In other words, at each moment in the replayed execution, the TTD framework only needs to ensure that application-visible objects have the same referential relationships. The replay system has the freedom to map those objects to arbitrary virtual memory locations.

2.2 Prior TTD Systems

Imagine that we desire to log and replay the following web program:

```
<script type="text/javascript">
window.addEventListener('load', function() {
  var callbackId = setInterval(function() {
    var box1 = document.getElementById('box1');
    var box2 = document.getElementById('box2');
    var box1Text = box1.innerText;
    box1.innerText = box2.innerText;
    box2.innerText = box1Text;
  }, 100);
});
</script>
<div id="box1">Now I'm Here!</div>
<div id="box2">Now I'm There!</div>
```

This program waits for the page to load, and then registers a timer that runs every 100 milliseconds and swaps the contents of the two `<div>` tags.

Suppose that we log and replay the program at the x86 layer. With this approach, the program has no external state, since there is no state that lives below the x86 layer. However, for a web developer who is only interested in application-level bugs, x86-level virtualization captures a large amount of extraneous internal state. For example, in a JavaScript program, the raw memory addresses of objects are extraneous state, since they cannot be observed by JavaScript programs. The raw contents of graphics memory are irrelevant, since the JavaScript runtime wraps low-level graphics hardware in the abstraction of the DOM [18]. Furthermore, all of the low-level memory layout is extraneous, since the JavaScript program only cares about the heap's referential integrity, not its raw pointer integrity. x86-level traces also record the content and timing of low-level hardware events like timer interrupts and IO interrupts. However, JavaScript programs only perceive the indirect side-effects of those events, through high-level interfaces like `setTimeout()` and `XMLHttpRequest`; thus, instruction-precise timings for hardware-level events are extraneous. Due to the large amount of extraneous state, program checkpoints can be very large, even with delta encoding [10, 45].

To reduce the overwhelming amount of extraneous state, a TTD system can interpose on the virtualization boundary defined by the managed runtime [5, 32]. At logging time, the TTD framework observes the output from each call to a managed runtime API; additionally, the framework wraps some inputs to API calls with code that assists with event logging. In the simple web program above, the TTD system logs the return value from `setInterval()`, and wraps the input function with code that logs the timer ID and the current time before invoking the program-provided function. At replay time, the TTD framework intercepts each call to a runtime API, and either directly returns a logged value, or allows the call to issue to the live, replay-time managed environment under the assumption that the interaction is deterministic. A traditional time-traveling debugger like Tardis [5] would do the former, but Mugshot [32] does the latter for modifications to DOM state like `box1.innerText`.

Unfortunately, both approaches are problematic. If the TTD framework only returns logged values for interactions with black-box state, then the replay-time debugger cannot query that state at arbitrary moments (§1), and the associated state cannot be recreated at replay time (e.g., a replayed web application will have no GUI state, only JavaScript state). Alternatively, if the TTD framework allows the replayed application to invoke APIs on live replay-time black boxes, the concurrently executing black boxes may race with the portion of the application that the debugger does control. These races can lead to erroneous replays (§3.2).

3 Gray-box Virtualization

A managed runtime presents applications with an interface to system resources like the network, the display, and local storage. Since the managed interface uses a high level of abstraction, some of the state which belongs to a program resides *beneath* the managed interface, inside the managed runtime itself. For example, when JavaScript code registers a timer via `setTimeout(callback, waitMs)`, the browser adds the two-tuple `<callback, invocationTime>` to a C++ queue of pending timers. The browser's managed runtime does not expose the timer queue to JavaScript code. However, a time-traveling debugger must log updates to the timer queue, so that timers can be dispatched at the appropriate moments during replay.

To expose such black-box state to debuggers, we use gray-box virtualization. Gray-box virtualization makes small modifications to the managed runtime to introduce *interrogative interfaces* that expose previously hidden application state. Gray-box virtualization tweaks the managed runtime to make application logging and checkpointing easier; this approach is similar in spirit to paravirtualization [4], which tweaks a guest OS to make it more amenable to x86-level virtualization. Note that our new interrogative interfaces are only exposed to debuggers, i.e., applications perceive no difference in how the managed runtime operates.

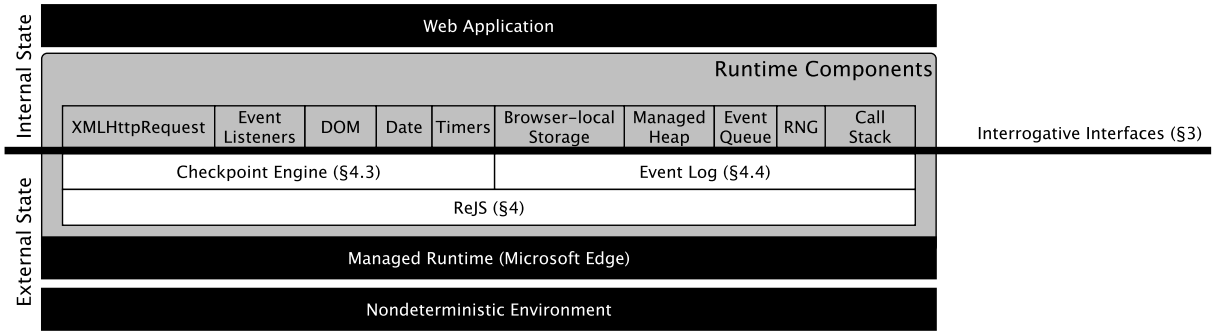

Figure 1: A system diagram of REJS. Program checkpoints and event logs only concern state from the shaded boxes, from which the runtime can derive OS and hardware state.

In the remainder of this section, we enumerate the interrogative interfaces that we added to a commercial-strength web browser (§3.1). We also discuss how interrogative interfaces allow debuggers to handle logging-time data races which cause replay-time problems for prior debuggers (§3.2).

3.1 Browser Implementation

Figure 1 illustrates the architecture of our modified web browser. We use Table 1 to frame our discussion of the implementation details.

Heap: In a managed runtime, the bulk of an application's state lives in the heap. Efficient heap snapshots are therefore a prerequisite for a variety of program analyses and debugging tools. Fortunately, managed languages already ship with fast garbage collectors that can walk the heap and discover live objects. We extend the browser's pre-existing garbage collector with an interrogative interface that can serialize the live portion of the current object graph. We add another interface that can reinflate a previously serialized heap.

Stack: A JavaScript application is completely event-driven. Thus, there is no application-level stack state between the dispatch of two events. If a debugger only generates checkpoints during the quiescent period between events, the debugger does not require interrogative methods for inspecting the stack. Deferring checkpoints until event handler termination does not unduly postpone the checkpoints, since an event handler typically only runs for a few milliseconds to keep the UI responsive [41].

RNG state: JavaScript applications use `Math.random()` to generate random numbers. However, applications cannot read or write the internal state of the PRNG. We add an interrogative interface that allows a debugger to query and reset the PRNG state. At logging time, the debugger can record the initial PRNG state; at replay time, the debugger can properly initialize the PRNG before allowing the application to execute.

Current time: The JavaScript runtime exposes a `Date` interface that lets a debugger observe the current time, which it queries from the OS directly. We add an interrogative interface that lets a debugger act as the clock source for the application. When the application requests the current time, the runtime consults the debugger, which can choose what value to return.

Timer status: JavaScript applications create one-shot timers via `setTimeout()`, and recurring timers via `setInterval()`. Each timer is assigned a unique ID that is determined arbitrarily by the runtime. The runtime does not provide a mechanism for applications to enumerate the set of active timers, their IDs, and their activation dates. We add an interrogative interface which exposes that information to a debugger.

GUI contents: JavaScript code uses the Document Object Model (DOM) to interact with the display [18]. Each HTML tag in a page has a corresponding element in the DOM tree (a JavaScript-visible data structure). Each DOM element provides access to tag-specific state, such as the URL for an `<img>` tag, or the rendered size of a `<div>` tag. However, the DOM interface does not expose the CSS animation state of an HTML tag—that state resides within the black-box renderer. If a debugger cannot read CSS animation tick counts, the debugger cannot record an animation's progression with respect to concurrently executing JavaScript code; this ignorance prevents the debugger from faithfully recreating the behavior at replay time. To enable high-fidelity replays of animations, we add an interrogative interface to the black box renderer, allowing a debugger to read and write the tick counts which belong to active CSS animations.

A web page often includes external objects, i.e., HTML tags which specify a `src` attribute and whose content must be fetched from remote servers. When the content

finishes loading, the browser silently updates the applicable DOM node with attributes, such as the `height` and `width` of an image. We add interrogative methods to the black-box network stack, allowing a debugger to log, inject, or suppress these fetches. The interrogative methods allow the debugger to recreate nondeterministic network events at replay time.

Event listeners: A JavaScript program can register handlers for other event types besides network events. For example, a program can register handlers for GUI events, or for message events which contain data from other iframes. Applications can register event listeners in three ways: through properties on HTML tags (e.g., `<div onclick="someFunction()">`), properties on the DOM elements (e.g., `div.onclick = someFunction;`), or through the `addEventListener()` DOM interface (e.g., `div.addEventListener('click', someFunction)`). Applications can enumerate event listeners which were registered using the first two approaches, since the listeners are simple properties of the associated DOM objects. However, the browser runtime does not allow JavaScript code to enumerate handlers which were registered via `addEventListener()`. Furthermore, the runtime dispatches events to event handlers in the order in which the handlers were registered, regardless of the registration technique employed. The runtime does not expose this order (which must be recreated at replay time).

In JavaScript, all objects which generate events implement the `EventTarget` interface. Using an interrogative extension to that interface, we allow debuggers to enumerate all event handler information that is associated with an `EventTarget`. Using the interrogative extension, a debugger can track handler orders at logging time. At replay time, the debugger can restore the necessary handler orders using the preexisting handler registration interfaces.

Pending events: Each JavaScript execution context is single-threaded and completely event-driven. An execution context iteratively dequeues the event at the head of its event queue, and executes any handlers that are associated with that event type. When the queue is exhausted, the thread will wait for a new event to arrive. While an event handler call chain is executing, the runtime can schedule additional events by appending them to the queue. The runtime does not expose the queue to JavaScript code, but queue state is important for replay because it determines the order in which events should be dispatched. So, we add an interrogative interface that allows a debugger to read and write the event queue.

Connection status: JavaScript applications communicate with remote servers via `XMLHttpRequest` objects. Each one encapsulates the state of a single HTTP connection. At logging time, the debugger can observe the state of each connection using existing methods on `XMLHttpRequest` objects. However, at replay time, the debugger needs a mechanism to recreate logged `XMLHttpRequests` without creating actual network connections. Using new interrogative interfaces, we allow the debugger to create `XMLHttpRequests` from scratch, and set their internal connection state to arbitrary values.

Storage: Web pages manage persistent local data using cookies and the `localStorage` interface [18, 17]. Both mechanisms export a key/value API. The browser creates a separate storage area for each origin,[2] and prevents different origins from accessing each other's data. The debugger has access to all of these origins from inside the runtime, and can manipulate them using the same interfaces that are exposed to regular applications.

Performance monitors: CPUs and managed runtimes define a wide variety of performance counters [19, 34]. Browsers do not expose these performance monitors to JavaScript code, and their values do not affect JavaScript-visible state. However, performance monitors are crucial for time-traveling debuggers, which require visibility into execution timings and branching activity (§4.2).

We extend the browser runtime with two performance monitors. The *branch trace store* contains the last branch that was taken by each function that is currently on the call stack. The *timestamp store* contains the timestamp of each function on the call stack. A timestamp is represented as a two-tuple of the function's call count since enabling performance monitoring and the number of basic blocks executed thus far in the function call.

For simplicity, we implemented the two stores by augmenting the browser's JavaScript interpreter. When a performance monitor is enabled, we disable the browser's JIT compiler, forcing JavaScript execution to use the interpreter. A time-traveling debugger only requires performance monitoring when a replayed execution nears a target line of interest (§4.1), so our design has minimal performance impact.

3.2 Preventing Data Races

In some cases, a managed runtime will update internal runtime state *in parallel* with the execution of the managed program. Such data races are normally benign, but they are problematic in the context of time-travel debugging. For example, a web browser executes CSS animations and network requests using OS-level threads that reside beneath the JavaScript virtualization boundary; this means that DOM modifications can occur at the same time that JavaScript code executes. Normally, such concurrent execution is unproblematic—when JavaScript

[2] A page's origin is a 3-tuple consisting of the protocol, hostname, and port in the page's URL.

code queries the DOM, the runtime retrieves a snapshot of the DOM at that instant in time. However, to perform accurate execution replay, a debugger must ensure that replay-time queries retrieve the DOM snapshot that would have been seen at the equivalent time in the *original* execution run. JavaScript replay tools like Mugshot [32] fail to guarantee this property, since those tools replay JavaScript interactions with the DOM without controlling for replay-time data races between JavaScript code and black-box components.

To make the problem concrete, suppose that a web page uses CSS to make a `<div>` tag periodically change its color. Further suppose that JavaScript code wishes to query the current color of the `<div>`:

```
<style type="text/css">
@keyframes color_change {
  from { background-color: blue; }
  to { background-color: red; }
}
.picker {
  animation: color_change 5s infinite alternate;
}
</style>
<script type="text/javascript">
var div = document.getElementById('picker');
var color = getComputedStyle(div).backgroundColor;
// Check if color is blue
if (color === 'rgb(0, 0, 255)'){
  x = 0;
} else{
  x = 1;
}
</script>
```

The browser updates the DOM in parallel with JavaScript execution, so the value of `color` in the JavaScript code is nondeterministic. If this nondeterminism is not faithfully recreated at replay time, subsequent program behavior may diverge from the logging-time behavior. In the example above, divergence would result in the replay-time program taking a different branch than the logging-time program, leading to a divergent assignment to `x`.

To eliminate these problems, gray-box virtualization constrains the times at which animation state and network state can change. At logging time, our interrogative code prevents animation threads and network threads from executing while JavaScript code executes. JavaScript code is single-threaded, so our policy results in animations and network events firing between the dispatch of JavaScript events. JavaScript event handlers are short in duration [41], so our policy does not result in user-perceived sluggishness of animations or network events. At replay time, the debugger uses log information to modify the GUI and replay network events before dispatching the next JavaScript event.

4 REJS

We build REJS, a time-traveling debugger for web applications, atop the gray-box virtualization support described in Section 3. REJS provides reverse-facing complements to existing debugger features (e.g. *step back* is the reverse of *step forward*). REJS uses interrogative interfaces to take periodic program checkpoints and to capture nondeterminism in an event log during program execution; Figure 1 displays an architectural diagram.

4.1 Time-Travel Overview

During normal program execution, i.e., before a time travel debugging session has started, REJS enables the event log and creates checkpoints at regular intervals. The checkpoint interval places an upper bound on the cost to seek to an arbitrary point in the program's execution.

To quickly time-travel an application to statement s at time t, REJS loads the last checkpoint taken before t and replays the event log. When execution is *close* to t, REJS enables the *branch trace store* and *timestamp store* via the performance monitoring interrogative interface and places a breakpoint on s conditioned on t. When there is no target time travel time, as is the case when the developer manually initializes replay from a particular snapshot, REJS must enable the performance monitors in order to respond to fine-grained time-travel requests at any given moment, e.g. "step back to the previous statement".

If the checkpoint interval is too large, jumping through time is slow, since REJS will replay many events to reach the desired execution point. On the other hand, if REJS records too many checkpoints during normal program execution, the program may suffer poor performance. At replay time, REJS can opportunistically generate checkpoints to reduce the latency of future time travel operations. In particular, if REJS is time traveling towards t, and must start from a "far-away" checkpoint (where distance is defined in terms of events), REJS generates a new checkpoint just prior to t. This optimization is motivated by common debugging scenarios in which developers explore a set of program states that are temporally clustered together. When a checkpoint exists that is close to t, returning to t is as fast as stepping forward to the next statement in a traditional debugger (§5).

4.2 Debugger Features

REJS provides a full suite of reverse-facing complements to existing debugger features. Due to space constraints, we only discuss *step back* in this section; the remaining features are implemented in a similar fashion.

Step back complements *step forward*, and lets the developer return to the previously-executed program statement. Given that the debugger is paused at the statement s at logical time $t = (c, b)$, where c is the number of times the function has been called since enabling performance monitoring and b is the number of basic blocks executed

in the current function call, the debugger must determine the statement and logical time of the previously-executed statement, s' and t':

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

Finally, REJS places a breakpoint on s' conditioned on t', and triggers replay from the previous checkpoint that is closest to the target logical time. If a checkpoint is not close to the target time, REJS deposits a new checkpoint just before the target JavaScript event.

4.3 Checkpoint Engine

Gray-box virtualization provides REJS with an omniscient view of the application's state. Using the interfaces described in Section 3.1, REJS can checkpoint the application's state into a snapshot on disk in a straightforward manner. The same interfaces let REJS restore the program to a previous state from a checkpoint.

4.4 Event Log

All of the nondeterminism in a web application stems from interactions with the state in Table 1. REJS uses the interrogative interfaces described in §3.1 to log nondeterministic updates to this state, and to keep the state in sync with JavaScript execution during replay.

In JavaScript, many interfaces in Table 1 are deterministic, and require no entries in the event log to replay. A few types of state need to be explicitly handled, which we discuss below.

Current Time: An interrogative interface lets REJS become the clock source for the application. REJS uses this interface to log clock values and to return logged values during replay.

Connection Status: REJS records updates to the program-observable fields on `XMLHttpRequest` objects into the log. During replay, REJS uses an interrogative interface to produce and update mock `XMLHttpRequest` objects that contain the recorded state, but do not open a network connection. Since `XMLHttpRequest` state updates become program-visible during quiescent points between JavaScript events, REJS can trivially apply them at the appropriate logical time during replay.

Timer Status: JavaScript timer interfaces are represented by a unique numeric ID, which the program uses as a token to cancel the timer. An interrogative interface lets REJS determine these IDs; REJS logs the IDs it issues, and replays them to the application. In addition, during replay, REJS prevents the runtime from triggering timer events itself as REJS replays the logged events from the original execution.

Pending Events: At log time, REJS records each event as it exits the event queue and executes. Each event is a tuple of the `EventTarget`, the target JavaScript function to invoke, and an event object that describes the event, which must be serialized into the log.

REJS tags each `EventTarget` object with a unique ID when the program first registers an event listener on it, and maintains a map from these IDs to objects. REJS represents the `EventTarget` object in the log using this ID. For the handler, REJS uses an interrogative interface to determine the handler's position in the corresponding handler list on the `EventTarget`, and represents the handler in the log as its position in the list. Finally, the event object itself contains static information that REJS can simply log as primitive values. During replay, REJS prevents the JavaScript runtime from inserting new events into the queue, and replays events from the log.

When an `EventTarget` is a DOM element in the GUI, REJS cannot guarantee that it has tagged the element prior to the event. The program can create DOM elements and attach JavaScript event listeners to it in HTML, which occurs without the JavaScript engine's cooperation. However, since the DOM element must be visible in the DOM tree, we can ID the element as its *tree path*, which is an array of numbers that uniquely identifies the path to the element in the DOM tree. At replay time, REJS can use the path to identify the correct element in the live DOM.

Timer events are an anomaly, and do not have an `EventTarget` object or an event object. Timers are represented using a unique numeric ID, which maps to exactly one JavaScript function that should be invoked when the timer fires. REJS logs these events using the timer's ID, which it has access to via an interrogative interface. During replay, REJS looks up the JavaScript function associated with the ID, and invokes it manually.

GUI Contents: Most interactions with the DOM are completely deterministic, but there are a few exceptions. Concurrent with JavaScript execution, components in the browser can silently update properties on DOM nodes in response to external entities, leading to nondeterminism that REJS must log (§3.2).

HTML elements can reference external network resources that the runtime downloads from the network. At log time, REJS uses an interrogative interface to *defer* these network events to a quiescent point between JavaScript events and log their contents. During replay, REJS uses the same interface to replay each network event to the native DOM component at the proper event

boundary.

CSS animations also occur in parallel with JavaScript execution, altering properties on animated DOM elements as they execute. REJS uses an interrogative interface to pause animations just before each JavaScript event and log their tick counts. When a JavaScript event finishes, REJS resumes the animation. During replay, REJS uses the same interrogative interface to restore animations to the appropriate tick counts prior to each event.

When a user interacts with an HTML form, such as by checking a checkbox or typing text into a textfield, the runtime updates the DOM. These interactions occur *only* at quiescent points between JavaScript events. REJS scans each form element prior to each JavaScript event, and logs any changes to their values. REJS uses this information to restore these values at the appropriate time during replay.

4.5 Program Migration

Since gray-box virtualization exposes the managed runtime's state at a high level of abstraction, REJS's checkpoints are *small*, *fast*, *complete*, and *portable*. These attributes enable efficient program migration over a wide area network; our benchmark applications can be migrated in *less than one second* end-to-end (§5.2). To support transitioning a checkpointed application into a "live" state, we handle certain bits of state from Table 1 slightly differently than we do during program replay.

Network Connections: Active network connections in the checkpoint, represented by `XMLHttpRequest` objects, are recreated using an interrogative interface and *transitioned* to a closed state to emulate a down network link. Web applications are programmed to be robust to these types of issues since users put their devices to sleep and disconnect from networks frequently, and should restart or relaunch failed requests.

Timer Status: Pending timers are recreated from the checkpoint, and are set to execute at the recorded trigger time. If the recorded trigger time has elapsed, the timer events are added to the event queue for immediate execution.

Pending Events: Pending events are re-inserted into the event queue via the interrogative interface, and are allowed to execute normally.

5 Evaluation

We benchmark REJS in multiple dimensions on a suite of unmodified programs and benchmarks. We ran the evaluation on a desktop with a quad-core Intel Xeon E5-1620 clocked at 3.6 GHz, 16GB of RAM, and a mechanical 7200 RPM SATA hard drive.

Our evaluation workloads represent a wide variety of browser applications, including computationally intensive JavaScript workloads, framework-heavy GUI programs, and event-heavy games:

- We use **Delta-Blue**, **Earley-Boyer**, **RayTrace**, and **Splay** from the *Octane* benchmark suite [37], which are all compute/memory intensive workloads. We modify the benchmarks to extend their runtime to $\sim$10 seconds to isolate REJS overhead from parsing/JIT warmup overhead.
- **RayTraceGUI** is the Octane RayTrace program with the original UI, which introduces nondeterministic DOM input to the deterministic benchmark [42].
- **ColorGame** [11] is an implementation of a test that demonstrates the *Stroop effect* [44]. It uses AngularJS and jQuery, which are both complicated and commonly used libraries, and result in ColorGame having $\sim 3\times$ as much code as the next largest benchmark. AngularJS exercises a wide variety of DOM features, and encodes crucial application data into the DOM directly.
- **CRUD** [12] is a standard content management interface that uses jQuery for all of its DOM interactions.
- **PacMan** [39] is an implementation of the classic Pac-Man game using the HTML5 canvas. It uses timers to update the contents of the canvas every 80ms, and stresses the checkpoint engine's ability to quickly serialize large DOM objects and REJS's ability to log large numbers of event callbacks.

Table 2 describes the code size of each of these benchmarks, including HTML documents and JavaScript libraries. Although ColorGame and CRUD are significantly larger than most of the other benchmarks due to their sizeable dependencies, they are representative of the complexity of existing applications on the web.

5.1 Checkpoint Engine Performance

To evaluate the performance of REJS's checkpoint engine, we run the benchmark programs in a configuration that checkpoints their state every second. For each benchmark, we calculate the average of the following metrics over all checkpoints:

- Time to produce the checkpoint (*Create Time*)
- Time to write checkpoint to disk (*Write Time*)
- Time to resume from checkpoint (*Inflate Time*)
- Size of checkpoint in memory (*Snap Memory*)
- Size of compressed checkpoint on disk (*File Size*)
- # of JavaScript objects in checkpoint (*JS Objs*)

In addition, we measure the maximum amount of virtual memory in use by the web browser process at checkpoints to illustrate the space required to represent each benchmark's state at the process level (*Process Size*).

Program	Code	Process Sz.	JS Objs	Create	Write	Inflate	Snap Mem.	File Sz.
Color-Game	746 KB	44 MB	18 K	0.12 s	0.12 s	0.43 s	3.3 MB	0.9 MB
CRUD	250 KB	28 MB	14 K	0.11 s	0.10 s	0.37 s	2.4 MB	0.5 MB
Delta-Blue	36 KB	34 MB	13 K	0.09 s	0.17 s	0.36 s	2.1 MB	0.4 MB
Earley-Boyer	244 KB	123 MB	17 K	0.09 s	0.11 s	0.30 s	3.3 MB	0.6 MB
PacMan	50 KB	31 MB	13 K	0.13 s	0.14 s	0.45 s	2.2 MB	0.5 MB
RayTrace	38 KB	73 MB	12 K	0.06 s	0.09 s	0.28 s	2.1 MB	0.4 MB
Splay	17 KB	538 MB	12 K	0.08 s	0.10 s	0.33 s	2.3 MB	0.5 MB

Table 2: The results from the checkpoint evaluation on the benchmark programs. It takes a fraction of a second to create or restore a checkpoint, and checkpoints are significantly smaller than the size of the browser process they capture.

Table 2 displays the results of these experiments. From our results, we make the following observations:

REJS's checkpoint operations are fast. REJS takes only a fraction of a second to completely checkpoint and serialize application state to disk, which is nearly imperceptible to users. The inverse operation, where REJS reads a checkpoint from disk and inflates it into memory, takes nearly the same amount of time.

In contrast, Hyper-V takes 3–5 seconds to snapshot an OS running REJS on our evaluation machine. In practice, nobody takes full, synchronous snapshots of heavyweight VMs, because such operations are known to be slow. Instead, people use various techniques to reduce snapshot costs, such as pre-migrating state ahead of the actual migration [10] and by using delta-encoding to create differential snapshots [45]. Our unoptimized prototype currently uses none of these techniques, yet is already fast enough for most applications.

Compressed REJS checkpoints are two orders of magnitude smaller than process-level memory footprints. REJS checkpoints contain only the JavaScript application's state in a high-density format that compresses to less than a megabyte. In contrast, the browser's in-memory layout is much more fragmented, contains application-irrelevant browser state, and is, on average, over 50 times larger than the in-memory representation of the program checkpoint, and over 240 times larger than the compressed checkpoint on disk. Hyper-V snapshots are even larger; on our evaluation machine, Hyper-V produces snapshots that are hundreds of megabytes large as they contain extraneous OS state.

5.2 End-to-End Program Migration

As discussed in Section 4.5, REJS's checkpoint engine can be repurposed for efficient program migration. For this scenario, we measure (1) the time required to extract and serialize the full program state (*Extract*), (2) the time required to send the state over a network (*Transmit*), and (3) the time to restore the program on a new machine (*Restore*). None of these operations occur in parallel; thus, the total amount of time required to migrate the program is the sum of these three numbers. We perform this experiment over a 10 Mbps hardwired ethernet connection.

Program	Extract	Transmit	Restore	Total
Color-Game	0.24 s	0.18 s	0.43 s	0.85 s
CRUD	0.21 s	0.10 s	0.37 s	0.58 s
PacMan	0.27 s	0.11 s	0.45 s	0.83 s
RayTraceGUI	0.20 s	0.08 s	0.38 s	0.66 s

Table 3: REJS takes *less than a second* to migrate a live browser application across a 10Mbps connection.

Table 3 shows the results from this experiment. It takes *less than a second* to migrate an application across the network. As expected, the extraction times and restoration times are consistent with the results in Table 2. Due to low checkpoint sizes, the transmission times are fractions of a second, and suggest that REJS can be used to efficiently migrate applications and buggy program contexts over bandwidth constrained networks.

5.3 Event Log Overhead

We evaluate the runtime and disk overhead of nondeterministic event logging on applicable benchmark applications.[3] We run each through a fixed series of events using a script to drive application inputs. Each script lasts approximately 10 seconds, and we run each benchmark with and without logging. For each benchmark, we measure the uncompressed log growth, and we compare the program runtime with and without logging to measure user-perceptible logging overhead. The results in Table 4 demonstrate that:

The event log grows at a glacially slow rate. The benchmark with the most nondeterminism, PacMan, has the largest low growth rate of 1.5KB/s. PacMan's primary source of nondeterminism is event scheduling related to

[3]The Octane benchmarks are deterministic, and are not applicable.

Program	Log Growth	Log Overhead
Color-Game	0.6 KB/s	†
CRUD	0.2 KB/s	†
PacMan	1.5 KB/s	†
RayTraceGUI	0.9 KB/s	†

Table 4: The results from the nondeterministic event log evaluation. The *uncompressed* log grows slowly and imposes no user perceptible slowdown (indicated by †).

Program	Overhead	TTS Startup	Step Time
Color-Game	4%	4.5 s	†
CRUD	†	3.2 s	†
PacMan	6%	4.7 s	†
RayTraceGUI	5%	2.9 s	†

Table 5: REJS imposes low overhead on our benchmark applications, performs *reverse step* as quickly as a traditional debugger performs *forward step* (indicated by †), and has an acceptable few-second startup time.

its 80 ms game loop. At that rate, REJS could record PacMan's execution for *over 11 years* on a 500GB hard drive. Note that reported log growth rates are *uncompressed*; with compression, that number would easily double.

Logging has no user-perceptible impact on program performance. Each benchmark took the same amount of time to execute with and without logging, which is unsurprising given that event logs grow at a slow rate.

5.4 Reverse Step Performance

Recall that REJS time-travels particular program states by replaying execution from the nearest checkpoint to the desired execution point. When a checkpoint is not located temporally close to the desired execution point, the developer observes a one-time *time-travel step startup* delay while the debugger replays execution and deposits a new checkpoint closer to the desired execution point. For this system, we evaluate (1) the slowdown encountered when executing programs in REJS (*Overhead*), (2) the time required to setup time-travel stepping (*TTS Startup*), and (3) the time required to take a single reverse step once time-travel stepping has started (*Step Time*). We drive the benchmark applications using the same scripts as the nondeterministic event log test, and in a configuration that checkpoints execution state every 2 seconds. To measure Step Time and TTS Startup, we chose 10 random breakpoints to reverse step from and took the average of the recorded metrics.

Table 5 shows the results from this evaluation. We make the following observations from these results:

Time-travel overhead is nearly imperceptible to the user. REJS imposes about 5% overhead on the benchmark applications during execution. In some cases, such as CRUD, REJS imposes *no* overhead because checkpoints occurs between events, when the application is waiting for user input. REJS can impose even lower runtime overheads in exchange for longer time-travel startup costs by lowering the checkpoint rate during execution.

REJS is as performant as a traditional debugger. After paying a one-time startup fee, the time to execute a *reverse step* in REJS is indistinguishable from executing a *forward step* in the browser's existing debugger.

Time-travel stepping startup costs are acceptable as a one-time cost. For randomly selected breakpoints, this startup cost is, on average, 3.8 seconds on our benchmark applications or roughly twice the checkpoint rate.

6 Discussion

We have only implemented gray-box virtualization for a web browser runtime. However, we believe that interrogative techniques can be applied to other managed runtimes like the JVM and CLR. Those runtimes operate at a lower level of abstraction than the browser runtime, and thus require a larger set of interrogative extensions. In this section, we explore some of the required extentions.

Thread scheduling and asynchronous events: A JavaScript execution context is single-threaded, but the JVM and CLR support multi-threaded execution contexts. To reproduce logging-time thread schedules at replay-time, the debugger can adopt a uniprocessor execution model, even if multiple cores are available [5, 23, 46, 43]. Alternatively, the debugger can use deterministic multi-threading to force predictable schedules [6, 13, 27, 38]. With either approach, we would add interrogative interfaces that allow the debugger to observe scheduling events and control when threads actually begin execution.

Checkpointing memory state: JavaScript's single-threaded, event-driven nature simplifies checkpointing, which can occur during the naturally quiescient periods between event dispatches. In the JVM and CLR, threads can execute indefinitely, without well-defined pause points. To force quiscience, we can leverage performance counters (§3.1) to track the execution progress of each thread. Once a thread has invoked a predefined number of functions, or triggered garbage-collected a predefined number of times, the runtime can pause the thread and vector control to the debugger. The debugger can wait for other threads to pause, and then checkpoint the entire application. A similar approach is used by Tardis, an existing time-traveling debugger for the CLR [5].

Application-supplied native methods: The JVM and CLR let programs embed functionality that is written in

unmanaged languages like C. If these native components are deterministic and reference no external state, they require no interrogative interfaces. Otherwise, the native components require interrogative extensions, similar to other black box components like a browser's rendering engine.

File system state: A web browser exposes the local disk using per-origin key/value stores. In contrast, JVM and CLR programs can directly access the local file system. To checkpoint file system state, a debugger can run applications atop a file system that already provides storage checkpoints [1, 2, 35]. Those file systems can also be network-mounted (e.g., via Amazon's EBS [3]), enabling "free" migration of file system state (since the destination machine can simply mount the remote volume that was used by the source machine).

Since the JVM and CLR expose the host file system, they allow applications to be affected by external, asynchronous modifications to the file system (similar to the `fcntl()` example from Section 1). Such modifications must be logged to ensure high-fidelity replays. Thus, we would need to extend the JVM and CLR to detect external file system events using mechanisms like inotify [21].

7 Related Work

Time-Travel Debugging and Deterministic Replay

As discussed in Sections 1 and 2.2, prior time-traveling debuggers are either precise and heavyweight, or imprecise but efficient. For example, x86-level hypervisors capture the entirety of a program's state, allowing for high-fidelity replay, but snapshots and logs contain extraneous information that is unnecessary for application-level debugging [14, 23, 47]. Tardis [5] and RR [43] mediate program interactions at a higher-level of abstraction (a managed runtime or OS interface, respectively). This approach gains smaller logs and more efficient replay, but loses the ability to faithfully track interactions with components like the GUI and the file system. REJS uses gray-box virtualization to faithfully and efficiently replay black-box components.

Mugshot [32] and Timelapse [7] interpose on the JavaScript event loop, and assume that replaying a logged sequence of mouse, keyboard, and network events will generate the same DOM states that were seen at logging time. Unfortunately, this assumption is not always true. As discussed in §3.2, the renderer and network stack can modify DOM state in parallel with the execution of JavaScript code. The resulting data races, if not properly logged, can result in replay divergence. REJS uses interrogative methods to properly capture interactions between the renderer, the network stack, and application JavaScript code.

Virtualization and Program Migration

A full snapshot for an x86-level VM is often a gigabyte or more, since large amounts of operating system state are bundled with application-level data. To make VM migration feasible (and bound the storage required for longitudinal snapshots), hypervisors use a variety of tricks to delta-encode snapshots and proactively push them to destination machines before a VM actually migrates [10, 45].

Container technologies like Docker [31] virtualize at a higher level of abstraction, but raw snapshots of Docker VMs are still hundreds of megabytes. By moving the the virtual machine boundary to the level of a managed runtime, we provide full, uncompressed checkpoints of *a few MB* that require a few tenths of a second to generate; checkpoint size and generation speed would improve even more if we used delta-encoding tricks.

Library operating systems migrate application-required OS state into the application's address space, removing extraneous OS state from application snapshots. For example, Drawbridge [40] provides a library implementation of a minimal win32 interface; Tardigrade [29] extends Drawbridge to provide fast VM checkpointing for replicated network services. However, full checkpoints are still tens to hundreds of megabytes large. Furthermore, adding interrogative methods to a managed runtime requires much less effort than refactoring a traditional OS into a library version.

The unikernel [30] approach compiles all of the software layers for an event-driven server into a single binary that runs atop raw hardware. Given modular kernel code, the unikernel approach results in VMs that are often less than a megabyte in size. However, OS services must be reimplemented as unikernel modules, and unikernels cannot support multithreading. In contrast, gray-box virtualization supports all programs that run on the target runtime, while providing VM checkpoints that are often as small as unikernel snapshots.

Imagen [28] is a migration system for the client-side state in a web application. Imagen uses source code rewriting and JavaScript's built-in reflection capabilities to manipulate application state. Imagen requires multiple seconds to checkpoint or resurrect a program. Imagen also relies on fragile shimming code to interpose on runtime interactions. Using robust gray-box virtualization, REJS does not need to rewrite programs, and can generate checkpoints an order of magnitude faster.

8 Conclusion

We describe a gray-box approach for high-performance and high-fidelity time-travel debugging. With *gray-box virtualization*, we extend components in the program runtime with *interrogative interfaces* to capture program in-

teractions missing from existing runtime APIs, making it possible to maintain live runtime state, such as the GUI, during time-travel. We implemented REJS, a time-traveling debugger for web applications, and show that REJS has less than 6% overhead during program recording, negligible overhead during *reverse step* operations, logs less than 1.5KB/s *uncompressed* on a wide variety of programs, keeps the GUI and other runtime state live during time-travel, and can migrate web applications in *less then one second* over a 10Mbps connection. REJS has been incorporated into Microsoft's open-source ChakraCore JavaScript engine [36].

References

[1] Btrfs Wiki. `https://btrfs.wiki.kernel.org/index.php/Main_Page`.

[2] OpenZFS. `http://open-zfs.org/wiki/Main_Page`.

[3] Amazon Web Services, Inc. Amazon EBS. `https://aws.amazon.com/ebs/`.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[5] E. T. Barr and M. Marron. Tardis: Affordable time-travel debugging in managed runtimes. In *OOPSLA*, 2014.

[6] M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. F. Salmi, S. Biswas, A. Sengupta, and J. Huang. OCTET: capturing and controlling cross-thread dependences efficiently. In *OOPSLA*, pages 693–712, 2013.

[7] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *UIST*, 2013.

[8] A. Burtsev, D. Johnson, M. Hibler, E. Eide, and J. Regehr. Abstractions for practical virtual machine replay. In *VEE*, 2016.

[9] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *ISSTA*, 2012.

[10] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *NSDI*, 2005.

[11] Angular-ColorGame. `https://github.com/vanessachem/Angular-ColorGame`.

[12] CRUD. `https://github.com/bersoriano/mobile-web-ui`.

[13] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *SOSP*, 2011.

[14] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *OSDI*, 2002.

[15] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 3rd edition, 1998.

[16] Z. Gu, E. T. Barr, D. Schleck, and Z. Su. Reusing debugging knowledge via trace-based bug search. In *OOPSLA*, 2012.

[17] I. Hickson. Web Storage (Second Edition). `http://www.w3.org/TR/webstorage/`, 2015.

[18] A. L. Hors, P. L. HÃl'garet, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document Object Model (DOM) Level 3 Core Specification. `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/`, 2004.

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

[20] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *ASE*, 2007.

[21] M. Kerrisk. Filesystem notification series by Michael Kerrisk. `https://lwn.net/Articles/605313/`.

[22] Y. P. Khoo, J. S. Foster, and M. Hicks. Expositor: scriptable time-travel debugging with first-class traces. In *ICSE*, pages 352–361, 2013.

[23] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX ATC*, 2005.

[24] A. J. Ko and B. A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *ICSE*, 2008.

[25] A. Lienhard, T. Gîrba, and O. Nierstrasz. Practical object-oriented back-in-time debugging. In *ECOOP*, 2008.

[26] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.

[27] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In *SOSP*, 2011.

[28] J. T. K. Lo, E. Wohlstadter, and A. Mesbah. Imagen: Runtime migration of browser sessions for JavaScript web applications. In *WWW*, 2013.

[29] J. R. Lorch, A. Baumann, L. Glendenning, D. T. Meyer, and A. Warfield. Tardigrade: Leveraging lightweight virtual machines to easily and efficiently construct fault-tolerant services. In *NSDI*, 2015.

[30] A. Madhavapeddy, R. Mortier, C. Rotsos, D. J. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: library operating systems for the cloud. In *ASPLOS*, 2013.

[31] D. Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239), Mar. 2014.

[32] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for Javascript applications. In *NSDI*, 2010.

[33] Microsoft. Common language runtime (clr). `https://msdn.microsoft.com/en-us/library/8bs2ecf4(v=vs.110).aspx`.

[34] Microsoft. PerformanceCounter Class. `https://msdn.microsoft.com/en-us/library/system.diagnostics.performancecounter(v=vs.110).aspx`.

[35] Microsoft. Resilient File System Overview. `https://technet.microsoft.com/en-us/library/hh831724.aspx`, 2013.

[36] Microsoft Corporation. Microsoft ChakraCore. `https://github.com/Microsoft/ChakraCore`.

[37] Octane v1. `https://developers.google.com/octane/`.

[38] M. Olszewski, J. Ansel, and S. P. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS*, 2009.

[39] Pacman. `https://github.com/bxia/Javascript-Pacman`.

[40] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library OS from the top down. In *ASPLOS*, 2011.

[41] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. Jsmeter: Comparing the behavior of javascript benchmarks with real web applications. In *USENIX Conference on Web Application Development, WebApps'10, Boston, Massachusetts, USA, June 23-24, 2010*, 2010.

[42] RayTraceGUI. `http://www.flog.co.nz/`.

[43] Mozilla rr tool. `http://rr-project.org/`.

[44] J. R. Stroop. Studies of interference in serial verbal reactions. *Journal of Experimental Psychology*, 18(6), 1935.

[45] P. Svärd, B. Hudzia, J. Tordsson, and E. Elmroth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. In *Proceedings of the 7th International Conference on Virtual Execution Environments, VEE 2011, Newport Beach, CA, USA, March 9-11, 2011 (co-located with ASPLOS 2011)*, pages 111–120, 2011.

[46] UndoDB v3.5. `http://undo-software.com`.

[47] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *MoBS*, 2007.