Rivet: Browser-agnostic Remote Debugging for Web Applications

James Mickens
Microsoft Research

Abstract

Rivet is the first fully-featured, browser-agnostic remote debugger for web applications. Using Rivet, developers can inspect and modify the state of live web pages that are running inside unmodified end-user web browsers. This allows developers to explore real application bugs in the context of the actual machines on which those bugs occur. To make an application Rivet-aware, developers simply add the Rivet JavaScript library to the client-side portion of the application. Later, when a user detects a problem with the application, the user informs Rivet; in turn, Rivet pauses the application and notifies a remote debug server that a debuggable session is available. The server can launch an interactive debugger front-end for a human developer, or use Rivet's live patching mechanism to automatically install a fix on the client or run diagnostics for offline analysis. Experiments show that Rivet imposes negligible overhead during normal application operation. At debug time, Rivet's network footprint is small, and Rivet is computationally fast enough to support non-trivial diagnostics and live patches.

1 Introduction

As an application becomes more complex, it inevitably accumulates more bugs, and a sophisticated debugging framework becomes invaluable for understanding the application's behavior. With modern web browsers providing increasingly powerful programming abstractions like threading [21] and bitmap rendering [9], web pages have become more complex, and thus more difficult to debug. Unfortunately, current debuggers for web applications have several limitations.

Although modern browsers ship with feature-rich JavaScript debuggers, these debuggers can typically only be used to examine pages running on the local browser. This makes it impossible for developers to examine real bugs from pages running on web browsers in the wild. A few browsers do support an interface for remote debugger attachment [4]; however, that interface is tied to a specific browser engine, meaning that the remote debugger cannot be used with other browser types. Given the empirical diversity of the browsers used in the wild [20], and the inability of web developers to dictate which browsers their users will employ, remote debugging frameworks that are tied to a particular browser engine will have poor coverage for exposing bugs in the wild. Indeed, the quirks of individual browsers are important inducers of web application bugs [13, 14]. Thus, an effective remote debugger must be able to inspect pages that run inside arbitrary, unmodified commodity browsers.

1.1 Our Solution: Rivet

In this paper, we introduce Rivet, a new framework for remotely debugging web applications. Rivet leverages JavaScript's built-in reflection capabilities to provide a browser-agnostic debugging system. To use Rivet, an application includes the Rivet JavaScript library. When the page loads, the Rivet library instruments the runtime, tracking the creation of various types of state that the application would otherwise be unable to explicitly enumerate. Later, if the user detects a problem with the web page, she can instruct the page to open a debugging session with a remote developer. The developer-side debugger communicates with the client-side Rivet framework using standard HTTP requests. The debugger is fully-featured and supports standard facilities like breakpoints, stack traces, and dynamic variable modification.

Many laypeople users will not be interested in assisting developers with long debugging sessions. Thus, Rivet also supports an automated diagnostic mode. In this mode, when the user or the application detects a problem, the debug server automatically pushes pre-generated test scripts to the client. The client executes the scripts and sends the results back to the server, where they can be analyzed later. In this fashion, Rivet supports the generation of quick error reports for live application deployments. This mechanism also facilitates the distribution of live patches to client machines.

1.2 Contributions

In summary, Rivet is the first fully-featured, browser-agnostic remote debugger for web applications. Rivet offers several advantages beyond its browser agnosticism. With respect to security, Rivet only allows a remote developer to debug pages from her origin; in contrast, prior (browser-specific) remote debuggers allow a developer to inspect any page in any browser tab. With respect to usability, Rivet does not require end-users to reconfigure their browsers or manage other client-side debugging infrastructure. This is important, since widely deployed applications are primarily used by non-technical laypeople who lack the sophistication to configure network ports or attach their browser to a remote debugging server. Prior remote debuggers require end users to perform such configuration tasks.

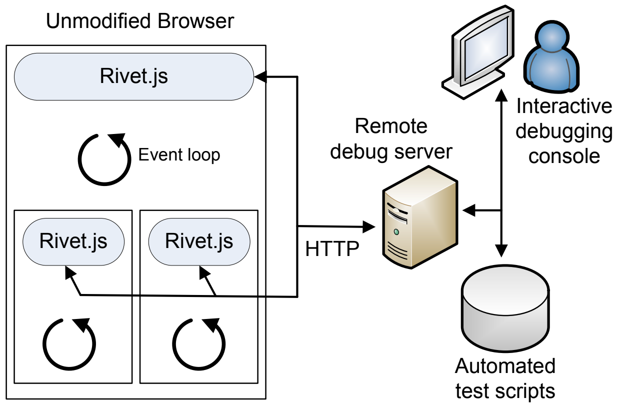

Figure 1: Example of a Rivet web page with three frames.

Rivet hides configuration details from the user by implementing its client-side component totally in JavaScript. This gives the web developer complete control over the client-side debugging settings, while restricting the developer to accessing content from that developer's origin. Rivet's client-side library is partially inspired by Mugshot [14], a logging-and-replay framework for web applications that also leverages JavaScript reflection to work on unmodified browsers. However, as we describe in Section 5, Rivet has three significant advantages over Mugshot. First, Rivet does not require developers to deploy a proxy web server to assist with application introspection; such proxies may be expensive in both cost and performance. Second, Rivet allows developers to explore bugs on the actual end-user browsers in which those bugs occur. This in-situ perspective increases Rivet's bug coverage relative to Mugshot—Mugshot tries to recreate bugs on the developer-side, but it cannot reliably replay bugs which depend on client-side state that resides beneath the JavaScript layer. Third, Rivet performs less runtime interpositioning than Mugshot. Since browser interpositioning can be fragile [13], this makes Rivet more robust than Mugshot.

Our evaluation shows that, through careful design, standard HTTP connections can support a fast, interactive remote debugging protocol. We also show that Rivet introduces negligible client-side overhead during normal application operation. Thus, Rivet is performant enough to ship inside real application deployments.

2 Architecture

Figure 1 depicts the architecture of a Rivet-enabled web application. The client portion of the application runs inside an unmodified commodity browser. The application consists of one or more *event contexts*. An event context is either an iframe or a web worker [21], a constrained type of threading context that we discuss in more detail below. JavaScript is an event-driven language, and each frame or web worker has its own event dispatch loop. Each context also has its own JavaScript namespace.

In a Rivet-enabled application, each event context contains its own copy of the Rivet library. When the application loads, each instance of the library will instrument its local JavaScript runtime. Later, at debug time, these modules will coordinate to pause the application, communicate with the remote debug server, and examine local state on behalf of that server.

On the developer side, the debug server may be connected to a front-end debugging GUI that allows the developer to interactively examine the client portion of the application. Alternatively, the developer can configure the debug server to run automated diagnostics on a misbehaving web page. The latter facility is useful for debugging widely deployed applications in which most users are not professional testers and lack an interest in actively assisting the developer in her debugging efforts. For a popular web page, the debug server may interface with a larger distributed system that collates and analyzes massive sets of error reports [10].

2.1 Exposing Application State

The purpose of the client-side Rivet library is to communicate with a remote debugger and give that debugger a way to inspect the client-side state. In particular, Rivet must allow the debugger to walk the JavaScript object graph that represents the application's state. Each event context has its own JavaScript namespace and possesses one or more partially overlapping object graphs. Below, we describe each type of object graph and how Rivet provides an introspection framework for that graph.

The global namespace forest: JavaScript supports a powerful reflection model. By calling the built-in `Object.getOwnPropertyNames()` method, an application can discover all of the properties defined on an object. [1] JavaScript also makes the global namespace explicitly accessible via the special `window` object. Thus, at debugging time, Rivet can discover the current set of global variables by enumerating the properties of the `window` variable. Rivet can then recursively reflect over the properties of each global variable, mapping the entire object tree rooted at each global.

The Document Object Model (DOM) is a browser-neutral API for exposing browser state to JavaScript programs [24]. With some notable exceptions that we

[1]JavaScript uses a prototype-based object model (§2.5), and `Object.getOwnPropertyNames()` only enumerates properties found directly on the object. Thus, to discover an object's full property list, Rivet must call `Object.getOwnPropertyNames()` on the object itself and on each object in the prototype chain.

```
function closureGenerator(obj){
    var y = 41;
    function closure(){
        //closure() binds to non-local
        //variables obj and y . . .
        return obj.x + y;
    }
    return closure;
}
var globalObj = {x: 1}; //Create object literal
var c = closureGenerator(globalObj);
alert(c()); //Displays 42
```

Figure 2: Example of a JavaScript closure.

```
function closureGenerator(obj){
    var y = 41;
    function closure(){
        return obj.x + y;
    }
    closure.__getScope__ = function(){
        return {"obj": obj, "y": y};
    };
    closure.__evalInScope__ = function(src){
        return eval(src);
    };
    return closure;
}
```

Figure 3: Rivet rewrites the closure from Figure 2 to support exploration by remote debuggers.

describe below, all DOM state is accessible via properties on the `window` object. For example, persistent local storage areas like cookies and DOM storage [25] are exposed via `window.document.cookie` and `window.localStorage`, respectively. For DOM state that is not explicitly accessible in this way, Rivet instruments the runtime to expose the state to remote debuggers.

Closure state: Many JavaScript applications make heavy use of closures. A closure is a function that remembers the values of non-local variables in the enclosing scopes. Figure 2 provides an example of a closure.

The global JavaScript namespace is explicitly bound to the `window` variable. In contrast, a closure namespace is *not* bound to an explicit namespace object. Thus, in Figure 2, once `closureGenerator(1)` returns, neither the returned closure nor any other code can explicitly refer to the bound closure scope—only the closure itself can access the bound variables `obj` and `y`. This is unfortunate from the perspective of debugging, since knowing the values of closure variables can greatly assist the debugging of closure functions. Some closure variables may be aliased by global variables, or reachable from the object graph rooted in a global variable; unfortunately, in non-trivial programs, it is difficult or impossible for a developer (or static code analysis) to identify such aliasing relationships. Furthermore, some closure variables are completely unreachable from the global namespace; in Figure 2, `y` is such a variable.

In JavaScript, all functions, including closures, are first-class objects. To expose closure state to remote debuggers, Rivet uses a JavaScript rewriter to associate each closure with two diagnostic functions that expose the normally hidden scope. Figure 3 shows how Rivet rewrites the closure from Figure 2. The `__getScope__()` function returns an object whose properties reference the closure variables. Note that `__getScope__()` is itself a closure, which lets it access the protected scope of the application-defined closure. When the remote debugger calls `__getScope__()`, it can use standard JavaScript reflection to enumerate the returned object's properties and read the closure variables.

Rivet adds a second diagnostic method to each closure called `__evalInScope__()`. This method allows a remote debugger to dynamically evaluate a piece of JavaScript code within the closure function's scope chain. This allows the debugger to modify the value of a closure variable (or any other variable within the function's scope). As we explain in Section 2.5, `__evalInScope__()` also allows the debugger to dynamically generate a new function that is bound to a closure's hidden scope. This is useful in several scenarios, e.g., when introducing a new debugging version of a closure that automatically produces diagnostic messages.

Note that `__getScope__()` and `__evalInScope__()` are only invoked during a debugging session, so rewritten closures incur no performance penalty during normal application execution. Also note that the closure rewriter can be implemented in a variety of ways. For example, a developer-side IDE can automatically rewrite JavaScript files. Alternatively, the client-side Rivet library can dynamically rewrite script code by interposing on `eval()` and DOM-mediated injection points for generating script code [15].

Importantly, Rivet will not break if some or all of an application's closures are not rewritten. However, Rivet will not be able to dynamically modify those closure functions. This will prevent Rivet from inserting dynamic breakpoints into them (§2.3) or otherwise live-patching them. Rivet can use lexical analysis to identify closures and prevent the developer from issuing forbidden operations on non-rewritten closures. This is an important feature, since an application may import external JavaScript that the developer does not control (and thus cannot ensure will be rewritten).

In JavaScript, calling a function's `toString()` method conveniently returns the source code for that

```
var button = document.getElementById("clicker");
button.onclick = function(){
    alert("DOM 0 handler!");
};
button.addEventListener("onclick",
                      function(){
                        alert("DOM 2 handler!");
                      });
```

Figure 4: Registering GUI callbacks using the DOM 0 model and the DOM 2 model.

```
var req = new XMLHttpRequest();
req.open('GET', "http://foo.com");
req.onReadyStateChange = function(){
    if(req.readyState == 4){
        alert(req.responseText);
    }
}
req.send();
```

Figure 5: An AJAX callback.

function. Thus, Rivet does not need to maintain extra metadata to store function source for subsequent inspection by the remote debugger.

Event handler state: JavaScript applications can define three types of callback functions.

- *Timer callbacks* execute after a specified period of time has elapsed. Callbacks registered via `setTimeout(f, waitMs)` only execute once, whereas callbacks registered via `setInterval(f, periodMs)` fire once every `periodMs` milliseconds.
- To respond to user input, applications define `GUI callbacks` on DOM nodes (each DOM node is the JavaScript representation of an HTML tag in the web page). Modern web browsers support two different registration models for GUI events [7]. The "DOM 0" model allows an application to register at most one handler for a given DOM node/event type pair. Using the "DOM 2" model, an application can register multiple event handlers for a given DOM node/event type pair. A node can simultaneously have a DOM 0 handler and one or more DOM 2 handlers. Figure 4 provides an example of the two registration models.
- To asynchronously fetch new web data, an application creates an `XMLHttpRequest` object and defines an `AJAX callback` for the object [7]. The browser will fire this callback whenever it receives bytes from the remote web server. Figure 5 provides an example.

The client-side Rivet framework must ensure that the remote debugger can enumerate the application-defined callbacks. Thus, when the Rivet library first loads, it wraps `setTimeout()`, `setInterval()`, and the AJAX object in logging code that records the application-defined handlers that are passed through the aforementioned interfaces. The wrapper code is similar to that used by the Mugshot logging and replay service [14]. Later, at debug time, the debugger can access the timer callbacks by examining the special lists `Rivet.timeoutCallbacks`, `Rivet.intervalCallbacks`, and `Rivet.AJAXCallbacks`.

Rivet does not need to do anything to expose DOM 0 handlers to the remote debugger—these handlers are attached to enumerable properties of the DOM nodes. In contrast, DOM 2 handlers are not discoverable by reflection, so Rivet modifies the class definition for DOM nodes, wrapping the `addEventListener()` function that is used to register DOM 2 handlers. The wrapper adds each newly registered handler to a list of functions associated with each DOM node; this list property, called `DOMNode.DOM2handlers`, is a regular field that can be accessed via standard JavaScript reflection.

Web Workers: A web application can launch a concurrent JavaScript activity using the web worker facility [21]. Although a web worker runs in parallel with the spawning context, it has several limitations. Unlike a traditional thread, a web worker does not share the namespace of its parent; instead, the parent and the child exchange pass-by-value strings over the asynchronous `postMessage()` channel. Web workers can generate AJAX requests and register timer callbacks, but they cannot interact with the DOM tree.

An application launches a web worker by creating a new `Worker` object. The application calls the object's `postMessage()` function to send data to the worker; the application receives data from the worker by registering DOM 0 or DOM 2 handlers for the worker's `message` event. At page load time, the Rivet library wraps the `Worker` class in logging code. Similar to Rivet's wrapper code for the `XMLHttpRequest` class, the `Worker` shim allows Rivet to track the extant instances of the class and the event handlers that have been added to those instances. Rivet also shims the AJAX class and the timer callback registration functions inside each worker context.

2.2 Pausing an Application

In the text above, we described how the client-side Rivet library exposes application state to the remote debugger. However, before the debugger can examine this state, the application must reach a *stable point*. A stable point occurs when Rivet-defined code is running within each client-side event context. Each frame or web worker can only execute a single callback at any given time, so if a Rivet-defined callback is running in a particular context, Rivet can be confident that no other application code is simultaneously running in that context. If Rivet code is running in *all* contexts, then Rivet has effectively paused the execution of all application-defined code. Rivet's client-side portion can then cooperate with the remote debugger to inspect and modify application state in an atomic fashion, without fear of concurrent updates issued by regular application code.

The top-most frame in a Rivet-enabled application is the coordinator for the pausing process. When the user detects a problem with the application, she informs the Rivet library in the root frame, e.g., by clicking a "panic" button that the developer has linked to the special `Rivet.pause()` function. `Rivet.pause()` causes the root frame to send a pause request to the Rivet library in each child frame and web worker; these pause requests are sent using the `postMessage()` JavaScript function. Upon receiving a pause command, a parent recursively forwards the command to its children. When a context with no children receives a pause request, it sends a pause confirmation to its parent and then opens a synchronous AJAX connection to the remote debug server. This synchronous connection allows the Rivet library to freeze the event dispatch process, forcing application-defined callbacks to queue up. A parent with children waits for confirmations from all of its children before notifying its own parent that it is paused and opening its own synchronous connection to the debug server. When the root frame receives pause confirmations from all of its children, it knows that the entire application is paused. The root frame connects to the remote debugger, which can then inspect the state of the application using the introspection facilities described in Section 2.1.

To unpause the application, the remote debugger sends a "terminate" message to each AJAX connection that it has established with the client. Upon receiving this message, each Rivet callback closes its AJAX connection and returns, unlocking the event loop and allowing the application to return to its normal execution mode.

2.3 Breakpoints

Rivet allows developers to dynamically insert breakpoints into a running application's event handlers. To set a breakpoint, the developer must first identify the appropriate event handler using the remote debugger's object enumeration GUI. This interface uses a standard tree view to represent the application's object graphs. Once the developer has found the appropriate function, the remote debugger displays the function's source code by calling the `toString()` of the underlying function object. The developer inserts breakpoints at one or more locations in the source code and then instructs the debugger to update the event handler function on the client side.

```
var lastEvalResult = "Start of breakpoint";
var exprToEval     = "";
while(lastEvalResult){
  exprToEval = Rivet.breakpoint(lastEvalResult);
        //Uses a synchronous AJAX connection
        //to return the result of the prior
        //debugger command and fetch a new one.
  lastEvalResult = eval(exprToEval);
}
```

Figure 6: The Rivet breakpoint implementation.

When the client-side Rivet framework receives the new handler source code, it translates each breakpoint into the code shown in Figure 6. Each breakpoint is just a loop which receives a command from the remote debugger, `eval()`s that command, and sends the result back to the debugger. A breakpoint command might fetch the value of a local variable or reset its value to something new. Communication with the debugger uses a synchronous AJAX connection to ensure that Rivet locks the event loop in the breakpoint's event context.

Once the client-side framework has translated the breakpoints, it dynamically creates a new function representing the instrumented event handler. To do so, Rivet passes the new source code to the standard `eval()` function if the handler to rewrite is not a closure; otherwise, Rivet uses the handler's `__evalInScope__()` function (§2.1). Equipped with the new callback, Rivet then replaces all references to the callback using the techniques we describe later in Section 2.5.

When the debugger unpauses the application, the application will execute normally until it hits a call to `Rivet.breakpoint()`. `Rivet.breakpoint()` must pause the application, but it must not relinquish control of the local event loop while it does so. Thus, `Rivet.breakpoint()` sends standard pause requests to its children, but does not wait for confirmations before marking itself as paused. `Rivet.breakpoint()` also sends a special "breakpoint" message to its parent. This message is recursively forwarded up the event context tree until it reaches the root frame. Upon receiving the message, the root frame pauses the rest of the application. Once the debugger has detected that all of the event contexts are paused, it can interrogate them as necessary.

2.4 Stack Traces

JavaScript does not explicitly expose function call stack frames to application-level code. Nevertheless, when an application hits a breakpoint, Rivet can send a stack trace to the remote debugger. Rivet defines two kinds of stack traces. A lightweight stack trace reports the function name and current line number for each active call frame. Even though Rivet does not have explicit access to the call stack, it can access function names and line numbers by intentionally generating an exception within a try/catch block, and extracting stack trace information from the native `Exception` object that the browser generates. This is the same technique used by the stacktrace.js library [23], and it works robustly across all modern browsers.

Lightweight stack traces do not expose actual call frames to Rivet. Thus, although Rivet breakpoints can use an `eval()` loop to provide read/write access to local variables in the topmost stack frame, Rivet cannot use lightweight stack traces to introspect variables in call frames that are lower in the stack. Rivet can provide heavyweight stack traces that provide such functionality, but to do so, Rivet must rewrite *all* functions so that they update a stack of function objects upon each call and return. Furthermore, each function must define the `__getScope__()` and `__evalInScope__()` that Rivet uses to introspect upon closure state (§2.1).

Both closure rewriting and heavyweight stack rewriting can be done statically, before application deployment. However, the function code that supports heavyweight stack traces can be dynamically swapped in at debug time. This allows the application to avoid the bookkeeping overhead during normal operation. Note that rewritten closure code should *always* be used, lest Rivet miss the creation of a closure scope and be unable to expose the bound variables to the remote debugger.

2.5 Generic Live Patching

Rivet defines a patch as a single JavaScript function which Rivet will evaluate in the global scope. The patch can access and modify the objects reachable from the global variables without assistance from Rivet. The patch accesses closure scopes through the `__getScope__()` function that Rivet adds to each closure. The patch can also access normally non-enumerable event handlers using data structures like `Rivet.timeoutCallbacks` and `DOMNode.DOM2handlers`.

To make patches easier to write, Rivet defines three convenience functions for developers. The first, called `Rivet.overwrite(oldObj, newObj)`, instructs Rivet to copy all of `newObj`'s values into `oldObj`. If neither `newObj` nor `oldObj` are a function, the overwriting process is trivial—since JavaScript objects are just dictionaries mapping property names to property values, Rivet can overwrite an object in place by deleting all of its old properties and assigning it all of `newObj`'s properties. If `newObj` or `oldObj` is a function (or another native code object like a regular expression), Rivet must use a different approach, since these objects are bound to opaque internal browser state that is not easily transferred to other objects. Thus, to overwrite a native object, Rivet must traverse the application's entire object graph and, for each object, replace any references to `oldObj` with references to `newObj`. Rivet uses standard techniques from garbage collection [12] to avoid infinite recursion when the object graph contains cycles.

```
//Define a constructor function for class X.
function X(){
    this.prop1 = 'one';
}
X.prototype.prop2 = 'two';

var x1 = new X();
var x2 = new X();
alert(x1.prop1); //'one': defined in constructor
alert(x1.prop2); //'two': defined by prototype

x1.prop1 = 'changed';
alert(x1.prop1 == x2.prop1);  //False

//Changing the value for a prototype
//property changes the value in all
//instances of that class *unless* an
//instance has explicitly redefined the
//property.
X.prototype.prop2 = 'cat';
alert(x1.prop2 == 'cat');  //True
x2.prop2 = 'dog';
alert(x2.prop2 == 'cat');  //False
```

Figure 7: Example of prototype-based inheritance.

The second convenience function, called `Rivet.redefineClass(oldCtor, makeNewVersion, newCtor)`, is useful for updating all instance objects of a modified class definition. Whereas C++ and Java implement classes using types, JavaScript implements classes using *prototype objects* [7]. Any JavaScript function can serve as a constructor if its invocation is preceded by the `new` operator; any references to `this` inside the constructor invocation will refer to the new object that is returned. Furthermore, by defining a `prototype` property for a function, an application defines an exemplar object which provides default properties for any instance of that constructor's objects. Figure 7 provides a simple example of JavaScript's prototype-based classes.

When a patch invokes `Rivet.redefineClass(oldCtor, makeNewVersion, newCtor)`, Rivet does the following:

- Rivet finds all instances of `oldCtor`'s class, i.e., all objects whose `__proto__` field is `oldCtor.prototype`.
- For each instance `obj`, Rivet calls `makeNewVersion(obj)`, creating a new

version of the object using application-specific logic. Rivet then update `oldObj` in-place with the contents of `newObj`.
- Finally, Rivet uses `Rivet.overwrite(oldCtor,newCtor)` to replace stale references to `oldCtor` with references to `newCtor`.

Applications that desire finer-grained control over their patching semantics can invoke `Rivet.traverseObjectGraph(callback)`, specifying a function that Rivet will call upon each object in the application.

2.6 Debugging Scenarios

We envision that Rivet will be used in two basic scenarios. In the beta testing scenario, a small number of professional testers or motivated volunteers interact with an unpolished version of an application. These testers are not developers themselves, but they are willing to start and stop their application sessions and actively help the developers to debug any problems that arise. In this scenario, when the beta tester encounters a problem, she initiates a full-blown debugging interaction with a remote developer. The developer starts the debugging GUI and engages in an iterative process with the beta tester and the remote application, exploring and modifying the program state in various ways. Using Rivet's live patching mechanism, the developer can dynamically add a chat interface to the live page; this enables a real-time dialogue between the tester and the developer without requiring the tester to configure a separate out-of-band communication mechanism.

For a widely deployed application, the typical layperson user may not want to assist with a remote debugging session. In these situations, the application can still contain a "panic" button. However, when this button is pressed, the application does not initiate a remote debugging session with a human developer—instead, the web page connects to an automated debug server which runs a set of predefined diagnostics on the page. Each diagnostic is implemented as a Rivet patch which simply runs a test on the client-side state and uploads the result to a server. For example, a patch might generate an application-level core dump, serializing the DOM tree and application heap and sending it to the server for further analysis. As another example, a diagnostic could run integrity checks over the application's cookies and other persistent client-side data. These kinds of automatic tests require no assistance from the end user, but provide invaluable debugging information to application developers. These tests can also be silently initiated by the Rivet library, e.g., in response to catching an unexpected exception. In some applications, such automatic diagnostics may be preferable to having a user-triggered "panic" button.

2.7 Implementation

Our Rivet prototype consists of a client-side JavaScript library and a developer-side debugging framework that contains a JavaScript rewriter, a debug server, and a front-end debugging GUI. After minification (i.e, the removal of comments and extraneous whitespace), the client-side library is less than 29 KB of source code; thus, it adds negligible cost to the intrinsic download penalty for a modern web application that contains hundreds of KB of JavaScript, CSS, and images [18]. The closure rewriter and the debug server are both written in Python. Before deploying an application, the developer passes its JavaScript code through the rewriter, which instruments closures as described in Section 2.1. After deploying the application, the debug server listens for debugging requests from remote web pages. If the server is configured to run in auto-response mode, it will run a pre-selected diagnostic script on the remote application and store the results in a database. Otherwise, if the server is set to interactive mode, it will open a front-end debugging GUI which the developer can use to inspect the remote application in real time. The front-end is just a web page that communicates with the debug server via HTTP. The server acts as a relay between the front-end and the remote application, sending debugger commands to the page and sending client-generated results back to the front-end.

Our current prototype implements all of the features described in this section except for heavyweight stack traces (§2.4). We are currently building a rewriting engine to support this facility. The client-side Rivet library has been tested extensively on Firefox, Safari, IE, and Chrome, and it works robustly on those browsers.

3 Privacy Concerns

Rivet exposes all of a web page's state to a remote debugger. This state might include personal information like passwords, emails, e-commerce shopping carts, and so on. Applications can use HTTPS to secure the connection between the web browser and the debug server; by doing so, a client can easily verify the identity of the remote principal that is inspecting local state. HTTPS also prevents arbitrary network snoopers from inspecting client data. However, it does not constrain the remote debugger's ability to enumerate and modify client-side state.

Rivet does not grant fundamentally new inspection powers to developers, since applications do not need Rivet to take advantage of JavaScript's powerful reflection capabilities. For example, there are many preexisting JavaScript libraries that analyze user activity and page state to determine which ads a user clicks or which parts of a page are accessed the most. Also, much of the private client-side information is persistently stored in

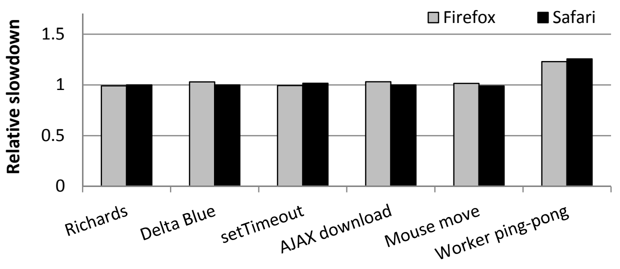

Figure 8: Microbenchmarks: Relative slowdown of Rivet-enabled versions. A slowdown factor of 1 indicates no slowdown.

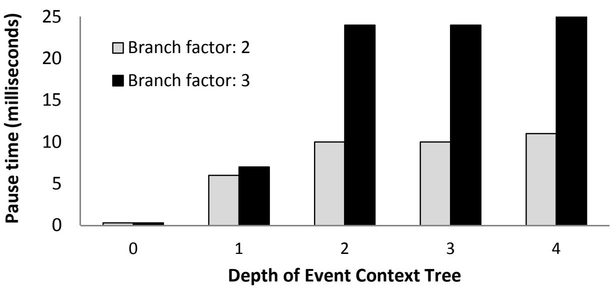

Figure 9: Time needed to pause a nested frame tree with no event loop contention.

server-side databases, with the client application treated as an ephemeral cache. Thus, developers can already inspect much of the user's private data without interrogating the client portion of the application.

4 Evaluation

In this section, we investigate four questions. First, how much overhead does Rivet add to an application during normal usage, i.e., when the application is *not* being debugged? Second, how long does it take for Rivet to pause an application? Third, how large are the messages that are exchanged between the client-side Rivet library and the debug server? Finally, how long does it take for Rivet to patch a live application or run an automated diagnostic? Using synthetic benchmarks and real applications, we show that Rivet adds negligible overhead during normal operation. Furthermore, at debug time, Rivet is fast enough to support interactive remote debugging sessions.

To test Rivet's performance on web applications in the wild, we loaded those applications through a rewriting web proxy. The proxy added the Rivet JavaScript library to each HTML file and web worker; it also passed all of the JavaScript code through Rivet's closure rewriter. Using this proxy, we could test Rivet's performance on live web applications without requiring control over the servers that actually deliver each application's content. In all graphs, each result is the average of ten trials. Each trial was run on a Dell workstation with dual 3.20 GHz processors, 4 GB of RAM, and the Windows 7 operating system. For the experiments in this section, we used Firefox 6.0.2 and Safari 5.1 to load Rivet-enabled applications. For graphs using only a single browser, Firefox was used.

4.1 Microbenchmarks

Computational slowdown: Figure 8 shows the relative performance slowdown for several Rivet-enabled microbenchmarks. We used the following benchmarks:

- `Richards` and `Delta Blue` are CPU-bound benchmarks from Google's V8 test suite.
- `setTimeout` measures how quickly the browser can dispatch timer callbacks that have a delay time of zero.
- The `AJAX download` test downloads a 100 KB file from a local web server on the same machine as the browser. Using a local web server eliminates the impact of external network conditions and isolates Rivet's impact on client-side AJAX handling.
- In the `Mouse move` test, the web page registers a null callback for mouse movement events. A human user then quickly moves the mouse cursor back and forth across the screen for ten seconds. The output of the benchmark is how many times the browser invoked the GUI callback.
- The `Worker ping-pong` test examines how quickly a web worker and a parent frame can send `postMessages()` to each other.

With the exception of the `Worker ping-pong` test, the performance of the Rivet-enabled benchmarks was statistically indistinguishable from that of the standard benchmarks. There are several reasons for Rivet's low overhead. First, the Rivet initialization code that runs during page load is extremely fast. This code merely needs to interpose on some class definitions and global methods, and this process takes less than one millisecond (which is the best resolution of JavaScript timestamps on modern browsers). Post-initialization, Rivet's bookkeeping code is also extremely fast. For example, the overhead of tracking `setTimeout()` callbacks is five lines of bookkeeping code and two extra function calls (one made from the wrapped `setTimeout()` to the native version, and one made from the wrapped callback to the real, application-supplied function). Rivet's overhead for tracking closure state is also low. As shown in Figure 3, rewritten closures do not execute any Rivet code during normal usage—the `__getScope__()` and `__evalInScope__()` functions are only invoked by the remote developer at debug time.

Figure 8 shows that Rivet makes the `Worker ping-pong` test roughly 25% slower. The primary reason is that browsers dispatch worker messages with much higher throughput than the rate at which they dispatch zero-delay timer callbacks or mouse events. As a concrete example, on our dual-core test machine, Firefox 6 could issue roughly 44 mouse events a second and 163 zero-delay timer callbacks a second, but over 16,000 ping-pong exchanges per second. Thus, Rivet's `postMessage()` overheads are comparatively larger. In a Rivet application, a single round of ping-pong involves three wrapped function calls: the `postMessage()` on the worker object in the parent context, the message callback in the worker context, and the message callback in the parent context that handles the worker's response.

Pause latency: A web application consists of a tree of event contexts. Rivet pauses an application by disseminating pause requests across this tree and waiting for child contexts to acknowledge that they are paused; the entire application is paused once the root frame has received pause confirmations from all of its children. Figure 9 depicts the pause latency as perceived by the root frame for event trees of various depths and branching factors. None of the event contexts defined any application-level handlers, so Rivet's pause handlers could execute as quickly as possible. Thus, the results in Figure 9 represent the lowest possible pause latencies.

Figure 9 shows that the intrinsic pause delay is extremely small. Even for an unrealistically dense tree with a branching factor of three and a depth of four, the pause process only takes 25 milliseconds. Of course, fully pausing an application can take an unbounded amount of time if an event handler contains code that runs for an extremely long time. However, we expect such situations to be rare, at least for handlers defined in frames, since developers know that long-running, non-yielding computations may freeze the browser's UI. Web workers were designed specifically so that applications could execute such computations without affecting the user interface; thus, web workers are a more likely source of long-running event handlers. However, in these situations, developers can explicitly insert Rivet breakpoints in worker code to place an upper-bound on an application's pause time.

Rivet's current pausing scheme guarantees that *all* contexts are in non-volatile states when debugging occurs. Rivet could trade this guarantee for the ability to diagnose hung contexts. At application load time, Rivet could create an invisible master frame that resided atop the context hierarchy. This master frame, controlled by Rivet, would always be guaranteed to be live. By coordinating with the Rivet libraries in each descendant event context, the master frame could build a list of all such contexts. Using this list, the master frame could send pause requests to each descendent context directly (instead of sending requests down the context tree). After a timeout, the master frame would declare any silent contexts to be hung. At that point, the master frame would yield to the remote debugger. The remote debugger could inspect both paused contexts and hung contexts as normal. However, *all* contexts would be in potentially volatile states, since a hung event handler could unhang at any moment and mutate some context's state before relinquishing the processor.

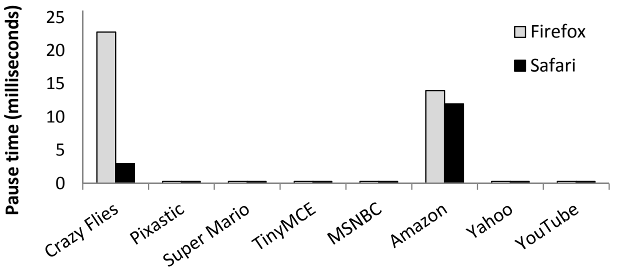

Figure 10: Time needed to pause several applications.

4.2 Macrobenchmarks

Pause latency: Figure 10 depicts the time needed to pause eight real applications. `Crazy Flies` simulates an insect swarm, using a web worker to calculate insect movements and a frame to depict an animation of the simulated insects' movements. `Pixastic` is an image manipulation program that supports standard operations like hue adjustment, noise removal, and edge detection. `Super Mario` is a JavaScript port of the popular 8-bit Nintendo game. `TinyMCE` is a full-featured word processor. The remaining web pages (`MSNBC`, `Amazon`, `Yahoo`, and `YouTube`) are the start pages for the associated web portals.

Figure 10 shows that in all but two cases, application pause times are essentially zero. This is because in most applications, the depth of the event context tree is either zero or one, and in each event context, there is little contention for the local event loop. The former means that the spanning tree for pause dissemination messages is small; the latter means that the Rivet library in each context does not need to wait long before it can grab the local event loop and pause the context. Thus, pausing is often very fast. Both `Crazy Flies` and `Amazon` had event context trees of depth one (`Crazy Flies` has a web worker, and `Amazon` has several frames). However, in both cases, pause times were less than 25 milliseconds.

Message sizes: Figure 11 depicts the size of the messages that Rivet generated during a debugging session for the `Yahoo` page. At the start of the session, the debug-

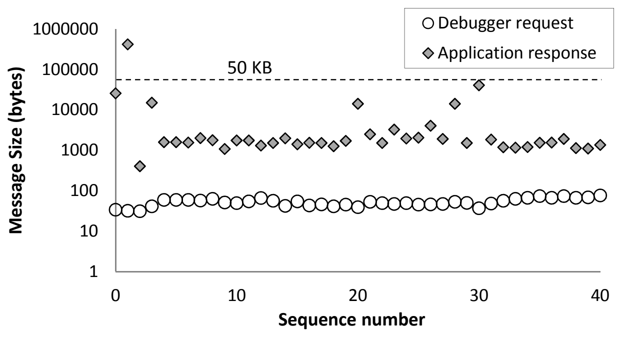

Figure 11: Size of messages exchanged between the remote debugger and the client application.

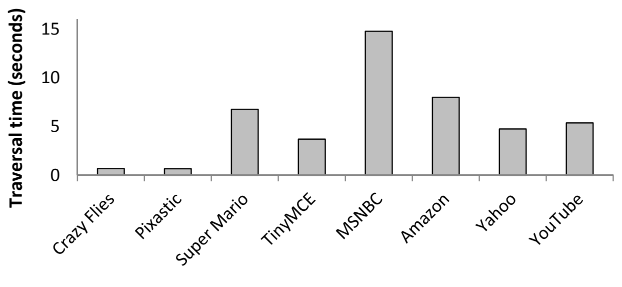

Figure 12: Time needed to visit every application object.

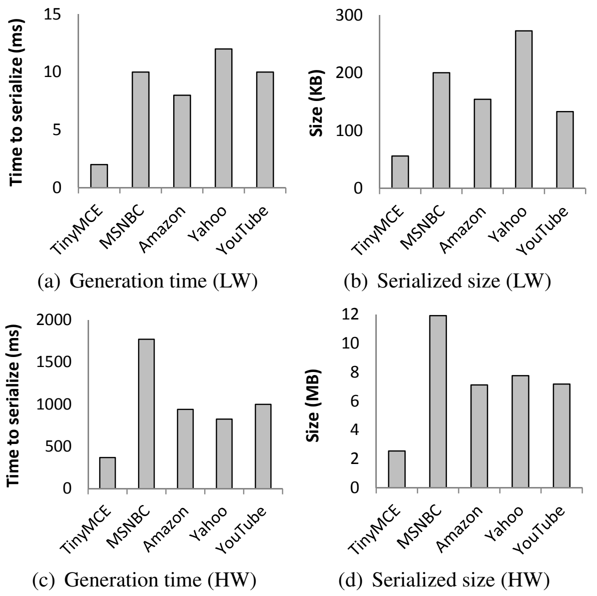

Figure 13: Computational overheads for lightweight (LW) and heavyweight (HW) DOM tree serialization.

ger automatically fetched the list of global variables and information about the children of the root `<html>` DOM node; this allowed the debugger GUI to populate the initial tree view elements that a human developer uses to explore the DOM tree and the object graph rooted by the `window` object. After this initialization completed, a human developer used the debugger GUI to explore various DOM nodes and application-defined objects.

Control messages from the remote debugger to the paused client application were extremely small, having an average size of 53 bytes and a maximum size of 74 bytes. The primary content of each control message was a list of property strings that indicated the path to the object whose contents should be returned. Compared to debugger requests, client responses were more variable in size, since objects had widely varying property counts. However, these messages were also small. The average client response was 13 KB, and all but one message was smaller than 50 KB. The outlier was the initial fetch of the global variable list and the properties of the associated objects. The browser defines over 200 built-in global variables, and a given application often adds ten or twenty more. Thus, the number of properties to fetch for the initial object view is often much larger than subsequent ones. Also note that for each function, the Rivet GUI fetches the associated source code by calling that function's `toString()`. This source code comprises the bulk of the initial view fetch.

Diagnostics: Once a web page is paused, a developer can run diagnostics on it or install a live patch. Rivet allows arbitrary dynamic code to be run on or inserted into the client-side. In this section, we discuss a few concrete examples of what a diagnostic might look like.

`Rivet.traverseObjectGraph(f)` allows the debugger to evaluate the function `f` over every object in an application. Thus, `Rivet.traverseObjectGraph()` is a useful foundation for many types of whole-application diagnostics. Figure 12 shows how long it takes Rivet to invoke a null function upon each object belonging to a particular application. This time is roughly linear in the number of objects that the application contains. In most cases, a full traversal only takes a few seconds. This means that a Rivet page supporting non-interactive, full graph diagnostics will only have to ask the user to keep a malfunctioning page open for a few seconds before the user can close it. Note that `MSNBC` has the largest traversal time (almost 15 seconds) because it has both a complex graph of application-defined JavaScript objects, and a complex, deep DOM tree. Although applications like `TinyMCE` and `Super Mario` also have large non-DOM object graphs, their DOM trees are comparatively simpler, leading to smaller traversal times.

Visual layout bugs are a common source of frustration in web applications [13]. Thus, developers will often be uninterested in viewing an application's entire object graph—instead, developers will be specifically interested in the application's DOM tree. Rivet provides built-in support for two types of DOM serialization. Lightweight serialization simply returns a string representing the page's HTML—Rivet easily generates this string by reading the `innerHTML` property of the page's `<html>` DOM node. The `innerHTML` string only reflects simple DOM properties like tag names and node ids, so it does not capture more complex application-defined properties like DOM 2 event handlers. Rivet's second type of serialization, which we call heavyweight serialization, does capture these properties. Heavyweight serialization essentially generates a serialized debugger view for the DOM tree. However, instead of generating view data for nested objects on demand, this data is recursively gathered in one sweep for analysis by the debug server.

Figure 13 depicts the computational requirements for lightweight and heavyweight DOM serialization. We ran the experiments on the five applications with the largest DOM trees. As expected, lightweight serialization was faster and generated smaller outputs than heavyweight serialization. The implementation of `innerHTML` resides completely within the native code of the browser, so reading the value only required a few milliseconds (Figure 13(a)). The resulting serialized strings were between 56 KB and 273 KB in size (Figure 13(b)). In contrast, generating the heavyweight serialization string was much slower, since the DOM tree traversal took place in application-level JavaScript code instead of browser-level C++ code. Figure 13(c) shows that heavyweight serialization took between 370 milliseconds and 1773 milliseconds. The serialization strings were also much larger, with the most complex sites like `MSNBC` and `Yahoo` having serialized DOM trees of 7–12 MB. On a modern DSL connection, such trees would take a few seconds to upload to the debug server. However, like the other client messages, we expect the serialized trees to be amenable to compression.

Live patching: Rivet supports live patching in addition to remote debugging. The installation speed of a live patch depends on the nature of that specific patch. For example, we created a patch that dynamically updated the `MSNBC` page to call an input sanitizer [15] whenever the user entered data into a text box; the input sanitizer stripped any dangerous HTML characters from the input, preventing cross-site scripting attacks [16]. The patch code used the built-in `document.getElementByTagName()` method to fetch all of the DOM nodes corresponding to text inputs. For each node, the patch installed a new handler for the `onchange` event that invoked the sanitizer.

Since `document.getElementByTagName()` is implemented by native browser code, this patch took only 11 milliseconds to execute. However, patches that are primarily implemented by application-level JavaScript will take longer to run. For example, we wrote a patch for `TinyMCE` that changed the definition of a class that represents URLs. The patch used the `Rivet.redefineClass()` method (§2.5) to perform the necessary modifications to the object graph. The resulting patch took almost eight seconds to install. This is because `Rivet.redefineClass()` has to make expensive application-level traversals through the object graph to replace stale object references.

Bug coverage: We have successfully used Rivet to explore known bugs in several in-house web applications; these bugs were caused by incorrect JavaScript code in the applications. Rivet can also explore the effects of bugs that arise from client-side configuration state that is not directly accessible to the JavaScript interpreter. For example, we used Rivet to reproduce a problem with the Firebug [6] plugin for Firefox, whereby enabling the plugin caused a severe performance decrease for the built-in JavaScript `eval()` function [5]. Mugshot [14], a JavaScript-level diagnostic framework like Rivet, cannot reliably examine bugs that involve client-side state beneath the JavaScript layer (§5).

5 Related Work

All modern browsers have built-in JavaScript debuggers. In most cases, these debuggers are only useful for examining the state of web pages running inside the local browser. The WebKit engine [22] used by Safari and Chrome does support remote debugging [4, 11, 17]. However, this debugging framework has three disadvantages. First, it only works for pages running inside Safari or Chrome. Second, the browser-side portion of the debugger must be manually configured by the end-user; for example, the user must decide which debug server should be allowed to connect to the local browser. Third, the debugging framework allows remote developers to inspect *any* page running on the remote browser, not just the ones that were created by the developer. In contrast, Rivet works on all browsers, requires no configuration work from end-users, and prevents a remote developer from inspecting pages that do not reside in her origin. Rivet also provides rich support for automated diagnostics, live patches, and real-time communication between end-users and remote developers.

There are browser-specific extensions for Firefox [1, 8] and the Android mobile browser [3] that add remote debugging facilities. Rivet has similar advantages over these systems—lack of end-user configuration activity, browser-agnosticism, and so on. However, because Rivet runs at the application level instead of inside the browser, Rivet cannot provide some of the low-level services that in-browser debuggers can provide. For example, Rivet cannot clear the browser cache or query the JavaScript garbage collector.

The JSConsole tool [19] provides remote access to a page's JavaScript logging console. JSConsole redefines the JavaScript `console` variable, replacing it with an object that implements the standard `console` interface but also accepts commands from a remote debugger. This allows a developer to inspect log messages generated by the application. The developer can also evaluate new JavaScript expressions within the context of the remote application. Like Rivet, JSConsole is browser-agnostic. However, compared to the debugging interface provided by Rivet or an in-browser debugger, the console interface is very limited. For example, JSConsole provides no way to set breakpoints, inspect closure state, or enumerate timer callbacks.

Mugshot [14] is a logging and replay framework for JavaScript web applications. Using Mugshot, a user who encounters a buggy application run can upload a log of the application's nondeterministic events to a remote developer. The developer can then replay the buggy execution run on a local browser, using the local browser's debugger to inspect the application's state.

Mugshot shares Rivet's goal of running on unmodified commodity browsers, and Rivet employs some of Mugshot's introspection techniques to instrument event contexts. However, Rivet interposes on fewer browser interfaces; this makes Rivet more robust and easier to maintain, since browser interpositioning is challenging to get correct [13, 14]. Mugshot also requires developers to deploy a special replay proxy that sits between the end user and the application web server. This proxy records the content and delivery order of client-requested information so that load order and load content can be faithfully recreated at replay time. Deploying this proxy may involve non-trivial effort, particularly if the application fetches content from external origins, since that content must be mirrored by the application's home servers so that it can be fetched through (and recorded by) the replay server. Rivet requires no such infrastructure. Rivet also has the advantage that it can examine application bugs in situ instead of having to transfer a log to the developer machine and recreate the application's state inside the developer's browser. This recreation process is somewhat fragile; for example, replay may lack fidelity if the developer does not select the same type of browser that the user has, or if the bug depends on client-side configuration state like DLLs that the client-side Mugshot library cannot see (and thus cannot describe to the remote developer). In contrast, Rivet runs inside the buggy application itself, and does not require state transferral or recreation. Rivet's interactive debugging mode allows developers to receive descriptions of local configuration state from the end-user.

Ksplice [2] is a system for live patching the Linux kernel. A Ksplice patch updates the kernel at the granularity of a function—to replace an old function, Ksplice adds the new function code to the kernel's address space and then inserts a jump instruction to the new code at the start of the old function. Rivet can perform similar tricks using JavaScript's built-in facilities for object reflection. Similar to Rivet's notion of a stable point, Ksplice defines a quiescent kernel function as one that is not on the call stack of any thread; only quiescent functions may be patched. If a Ksplice patch changes the layout of data structures, the developer must provide code to change these structures. Ksplice ensures that the execution of this code takes place within the same atomic transaction that updates functions. This process is similar to Rivet's mechanism for updating object definitions.

6 Conclusions

Rivet is the first browser-agnostic remote debugger for web applications. Rivet works on unmodified commodity browsers, taking advantage of JavaScript's intrinsic capabilities for dynamic object reflection and modification. Rivet's client-side consists of a JavaScript library; this library adds debugging hooks to the application and communicates with a remote debug server. Upon application failure, the remote server can initiate an interactive debugging session with the remote developer, or run automatic diagnostic scripts that produce data for offline analysis.

Experiments show that Rivet adds negligible overhead during standard application operation. At debug time, Rivet can pause an application in tens of milliseconds; subsequent debugging traffic between the application and the debug server is quite small, with debugger-to-application messages being 53 bytes on average, and application-to-debugger messages being 13 KB on average. Experiments also show that Rivet can efficiently support non-trivial diagnostics and live patches.

References

[1] ActiveState Software. Debugging JavaScript: Komodo 4.4 Documentation. `http://docs.activestate.com/komodo/4.4/debugjs.html`, 2011.

[2] J. Arnold and M. F. Kaashoek. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of EuroSys*, Nuremberg, Germany, April 2009.

[3] H. Correia. Remote JavaScript Debugging on Android. `http://www.sencha.com/blog/`

remote-javascript-debugging, November 5 2010.

[4] P. Feldman. WebKit Remote Debugging. http://www.webkit.org/blog/1620/webkit-remote-debugging/, May 9 2011.

[5] Firebug development site. Issue 732: eval() extremely slow when Firebug enabled. http://code.google.com/p/fbug/issues/detail?id=732, November 28 2011.

[6] Firebug: Web Development Evolved. http://getfirebug.com/, 2011.

[7] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., 5th edition, 2006.

[8] A. Fritze. JSSh: A TCP/IP JavaScript Shell Server for Mozilla. http://croczilla.com/bits_and_pieces/jssh/, 2009.

[9] S. Fulton and J. Fulton. *HTML5 Canvas*. O'Reilly Media, Inc., 1st edition, 2011.

[10] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *Proceedings of SOSP*, Big Sky, Montana, October 2009.

[11] Google. Chrome Developer Tools: Remote Debugging. http://code.google.com/chrome/devtools/docs/remote-debugging.html, 2011.

[12] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[13] J. Mickens and M. Dhawan. Atlantis: Robust, Extensible Execution Environments for Web Applications. In *Proceedings of SOSP*, Lisbon, Portugal, October 2011.

[14] J. Mickens, J. Howell, and J. Elson. Mugshot: Deterministic Capture and Replay for JavaScript Applications. In *Proceedings of NSDI*, San Jose, CA, April 2010.

[15] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. Draft specification, January 15, 2008.

[16] National Vulnerability Database. CVE-2010-2301, 2010. Cross-site scripting vulnerability: innerHTML.

[17] PhoneGap. Weinre: Web Inspector Remote. http://phonegap.github.com/weinre/, 2011.

[18] S. Ramachandran. Web metrics: Size and number of resources. Google. http://code.google.com/speed/articles/web-metrics.html, May 26 2010.

[19] R. Sharp. Remotely debug a mobile web app. http://www.jsconsole.com/remote-debugging.html, 2011.

[20] StatCounter. Global Stats: Top 5 Browsers from Sept. 2010 to Sept. 2011. http://gs.statcounter.com/, 2011.

[21] Web Hypertext Application Technology Working Group (WHATWG). Web Applications 1.0 (Living Standard): Web workers. http://www.whatwg.org/specs/web-apps/current-work/complete/workers.html, September 30 2011.

[22] WebKit Open Source Project. http://www.webkit.org/, 2011.

[23] E. Wendelin. stacktrace.js: A framework-agnostic, micro-library for getting stack traces in all web browsers. http://stacktracejs.org/, 2011.

[24] World Wide Web Consortium. Document Object Model (DOM). http://www.w3.org/DOM/, January 19 2005.

[25] World Wide Web Consortium: Web Apps Working Group. Web Storage: W3C Working Draft. http://www.w3.org/TR/2009/WD-webstorage-20091029, October 29 2009.