Atlantis: Robust, Extensible Execution Environments for Web Applications

James Mickens
Microsoft Research
mickens@microsoft.com

Mohan Dhawan
Rutgers University
mdhawan@cs.rutgers.edu

ABSTRACT
Today's web applications run inside a complex browser environment that is buggy, ill-specified, and implemented in different ways by different browsers. Thus, web applications that desire robustness must use a variety of conditional code paths and ugly hacks to deal with the vagaries of their runtime. Our new exokernel browser, called Atlantis, solves this problem by providing pages with an extensible execution environment. Atlantis defines a narrow API for basic services like collecting user input, exchanging network data, and rendering images. By composing these primitives, web pages can define custom, high-level execution environments. Thus, an application which does not want a dependence on Atlantis' predefined web stack can selectively redefine components of that stack, or define markup formats and scripting languages that look nothing like the current browser runtime. Unlike prior microkernel browsers like OP, and unlike compile-to-JavaScript frameworks like GWT, Atlantis is the first browsing system to truly minimize a web page's dependence on black box browser code. This makes it much easier to develop robust, secure web applications.

Categories and Subject Descriptors
D.4.7 [**Operating Systems**]: Organization and Design; D.4.5 [**Operating Systems**]: Reliability; D.4.6 [**Operating Systems**]: Security and Protection

General Terms
Design, Reliability, Security

Keywords
Web browsers, microkernels, exokernels

1. INTRODUCTION
Modern web browsers have evolved into sophisticated computational platforms. Unfortunately, creating robust web applications is challenging due to well-known quirks and deficiencies in commodity browsers [41]. In theory, there are a variety of standards that define the software stack for web applications [14, 19, 36, 52, 55]. In practice, the aggregate specification is so complex that it is difficult for any browser to implement it properly. To further complicate matters, browser vendors often add new features without consulting other vendors. Thus, each browser defines an idiosyncratic JavaScript interpreter, HTML parser, and layout engine.

These components are loosely compatible across different vendors, but fine-tuning an application for multiple browsers often means developing specialized application code intended for just one browser. For example:

- Writing a portable web-based GUI is challenging because different browsers handle mouse and keyboard events in different, buggy ways (§2.4.1).
- Web pages use CSS [52] to express complex visual styles. Some browsers do not support certain CSS elements, or do not implement them correctly (§2.4.3). When this happens, web developers must trick each browser into providing the proper layout by cobbling together CSS elements that the browser does understand.
- Browsers expose certain internal data structures as JavaScript objects that web pages can access. By introspecting these objects, pages can implement useful low-level services like logging/replay frameworks [31]. However, the reflection interface for internal browser objects is extremely brittle, making it challenging to implement low-level services in a reliable, portable fashion (§2.4.4).

All of these issues make it difficult for web developers to reason about the robustness and the security of their applications. JavaScript frameworks like jQuery [5] try to encapsulate browser incompatibilities, providing high-level services like GUI libraries atop an abstraction layer that hides conditional code paths. However, these frameworks cannot hide all browser bugs, or change the fundamental reality that some browsers support useful features that other browsers lack [27, 31]. Indeed, the abstraction libraries themselves may perform differently on different browsers due to unexpected incompatibilities [18, 28].

1.1 A Problem of Interface
These problems are problems of interface: web applications interact with the browser using a bloated, complex API that is hard to secure, difficult to implement correctly, and which

often exposes important functionality in an obscure way, if at all. For example, a page's visual appearance depends on how the page's HTML and CSS are parsed, how the parsed output is translated into a DOM tree (§2.2), how the resulting DOM tree is geometrically laid out, and how the renderer draws that layout. A web page cannot directly interact with any step of this process. Instead, the page can only provide HTML and CSS to the browser and hope that the browser behaves in the intended fashion. Unfortunately, this is not guaranteed [15, 30, 39, 41]. Using an abstraction framework like jQuery does not eliminate the fundamentally black box nature of the browser's rendering engine.

Using Xax [13] or NaCl [56], developers can write native code web applications, regaining low-level control over the execution environment and unlocking the performance that results from running on the bare metal. However, most web developers who are proficient with JavaScript, HTML, and CSS do not want to learn a native code development environment. Instead, these developers want a more robust version of their current software stack, a stack which does have several advantages over native code, like ease of development and the ability to deploy to any device which runs a browser.

1.2 Our Solution: Atlantis

To address these issues, we have created a new microkernel web browser called Atlantis. Atlantis' design was guided by a fundamental insight: the modern web stack is too complicated to be implemented by any browser in a robust, secure way that satisfies all web pages. The Atlantis kernel only defines a narrow, low-level API that provides basic services like network I/O, screen rendering, and the execution of abstract syntax trees [1] that respect same-origin security policies. Each web page composes these basic services to define a richer high-level runtime that is completely controlled by that page.

The Atlantis kernel places few restrictions on such a runtime. We envision that in many cases, web pages will customize a third-party implementation of the current HTML/CSS stack that was written in pure JavaScript and compiled to Atlantis ASTs. However, a page is free to use a different combination of scripting and markup technologies. Atlantis is agnostic about the details of the application stack—Atlantis' main role is to enforce the same origin policy and provide fair allocation of low-level system resources. Thus, Atlantis provides web developers with an unprecedented ability to customize the runtime environment for their pages. For example:

- Writing robust GUIs is easy because the developer has access to low-level input events, and can completely define higher-level event semantics.
- Creating a specific visual layout is straightforward because the developer can define his own HTML and CSS parsers, and use Atlantis' bitmap rendering APIs to precisely control how content is displayed.
- Pages can now safely implement low-level services that introspect "internal" browser state. The introspection is robust because the internal browser state is completely managed by the page itself—the Atlantis kernel knows nothing about traditional browser data structures like the DOM tree.

Atlantis' extensibility and complete agnosticism about the application stack differentiates it from prior browsers like OP [25] and IBOS [46] that also use a small kernel. Those systems are tightly coupled to the current browser abstractions for the web stack. For example, OP pushes the JavaScript interpreter and the HTML renderer outside the kernel, isolating them in separate processes connected by a message passing interface. However, the DOM tree abstraction is still managed by native code that a web page cannot introspect or modify in a principled way. In Atlantis, applications do not have to take dependencies on such opaque code bases. Thus, in contrast to prior microkernel browsers, Atlantis is more accurately described as an exokernel browser [16] in which web pages supply their own "library OSes" that implement the bulk of the web stack.

1.3 Contributions

Atlantis' primary contribution is to show that exokernel principles can lead to a browser that is not just more *secure*, but that also provides a more *extensible* execution environment; in turn, this increased extensibility allows web pages to be more *robust*. We provide a demonstration web stack that is written in pure JavaScript and contains an HTML parser, a layout engine, a DOM tree, and so on. This stack takes advantage of our new scripting engine, called Syphon, which provides several language features that make it easier to create application-defined runtimes (§3.3). We show that our prototype Syphon engine is fast enough to execute application-defined layout engines and GUI event handlers. We also demonstrate how easy it is to extend our demonstration web stack. For example, we show that it is trivial to modify the DOM tree `innerHTML` feature so that a sanitizer [33] is automatically invoked on write accesses. This allows a page to prevent script injection attacks [38].

The rest of this paper is organized as follows. In Section 2, we describe modern browser architectures, explaining why current browsers, both monolithic and microkernel, impede application robustness by exposing brittle black box and grey box interfaces. This discussion motivates the exokernel design of Atlantis, which we describe in Section 3. After describing our prototype implementation (§4) and several practical deployment issues (§5), we evaluate our prototype in Section 6, demonstrating its extensibility and its performance on a variety of tasks. We then discuss related work in Section 7 before concluding.

2. BACKGROUND

The vast majority of lay people, and many computer scientists, are unaware of the indignities that web developers currently face as they try to create stable, portable web applications. In this section, we describe the architecture of a modern web browser, and explain why even new research browsers fail to address the fundamental deficiency of the web programming model, namely, that the aggregate "web API" is too complex for any browser to implement in a robust fashion.

2.1 Core Web Technologies

Modern web development uses four essential technologies: HTML, CSS, JavaScript, and plugins. HTML [55] is a declarative markup language that describes the basic content in a web page. HTML defines a variety of tags for including different kinds of data; for example, an `<img>` tag references an external image, and a `<b>` tag indicates a section of bold text. Tags nest using an acyclic parent-child structure. Thus, a page's tags form a tree which is rooted by a top-level `<html>` node.

Using tags like `<b>`, HTML supports rudimentary manipulation of a page's visual appearance. However, cascading style sheets (often abbreviated as CSS [52]) provide much richer control. Using CSS, a page can choose fonts and color schemes, and specify how tags should be visually positioned with respect to each other.

JavaScript [20] is the most popular language for client-side browser scripting. JavaScript allows web pages to dynamically modify their HTML structure and register handlers for GUI events. JavaScript also allows a page to asynchronously fetch new data from web servers.

JavaScript has traditionally lacked access to client-side hardware like web cameras and microphones. However, a variety of native code plugins like Flash and Silverlight provide access to such resources. These plugins run within the browser's address space and are often used to manipulate audio or video data. A web page instantiates a plugin using a special HTML tag like `<object>`.

2.2 Standard Browser Modules

Browsers implement the core web technologies using a standard set of software components. The idiosyncratic experience of "surfing the web" on a particular browser is largely governed by how the browser implements these standard modules.

- The *network stack* implements various transfer protocols like `http://`, `https://`, `file://`, and so on.
- The *HTML parser* validates a page's HTML. The *CSS parser* performs a similar role for CSS. Since malformed HTML and CSS are pervasive, parsers define rules for coercing ill-specified pages into a valid format.
- The browser internally represents the HTML tag tree using a data structure called the *DOM tree*. "DOM" is an abbreviation for the Document Object Model, a browser-neutral standard for describing HTML content [51]. The DOM tree contains a node for every HTML tag. Each node is adorned with the associated CSS data, as well as any application-defined event handlers for GUI activity.
- The *layout and rendering engine* traverses the DOM tree and determines the visual size and spatial positioning of each element. For complex web pages, the layout engine may require multiple passes over the DOM tree to calculate the associated layout.
- The *JavaScript interpreter* provides two services. First, it implements the core JavaScript runtime. The core runtime defines basic datatypes like strings, and provides simple library services like random number generation. Second, the interpreter reflects the DOM tree into the JavaScript namespace, defining JavaScript objects which are essentially proxies for internal browser objects. These internal objects are written in *native code* (typically C++). From the perspective of a web application, the JavaScript wrappers for internal native objects should support the same programming semantics that are supported by application-defined objects. However, as we discuss later, browsers do not provide this equivalence in practice.
- The *storage layer* manages access to persistent data like cookies, cached web objects, and DOM storage [48], a new abstraction that provides each domain with several megabytes of key/value storage.

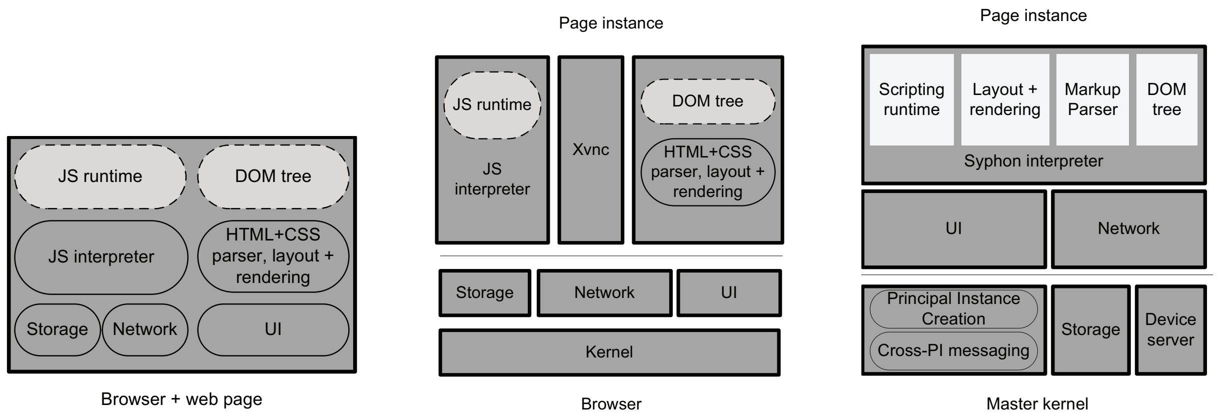
Even simple browser activities require a flurry of communication between the modules described above. For example, suppose that JavaScript code wants to dynamically add an image to a page. First, the JavaScript interpreter must send a fetch request to the network stack. Once the stack has fetched the image, it examines the response headers and caches the image if appropriate. The browser adds a new image node to the DOM tree, recalculates the layout, and renders the result. The updated DOM tree is then reflected into the JavaScript namespace, and the interpreter triggers any application-defined event handlers that are associated with the image load.

2.3 Isolating Browser Components

Figure 1(a) shows the architecture of a monolithic browser like Firefox or IE8. Monolithic browsers share two important characteristics. First, a browser "instance" consists of a process containing all of the components mentioned in Section 2.2. In some monolithic browsers, separate tabs receive separate processes; however, within a tab, browser components are not isolated.

The second characteristic of a monolithic browser is that, from the web page's perspective, all of the browser components are either black box or grey box. In particular, the HTML/CSS parser, layout engine, and renderer are all black boxes—the application has no way to monitor or directly influence the operation of these components. Instead, the application provides HTML and CSS as inputs, and receives a DOM tree and a screen repaint as outputs. The JavaScript runtime is grey box, since the JavaScript language provides powerful facilities for reflection and dynamic object modification. However, many important data structures are defined by native objects, and the JavaScript proxies for these objects are only partially compatible with JavaScript's ostensible object semantics. The reason is that these proxies are bound to hidden browser state that an application cannot directly observe. Thus, seemingly innocuous interactions with native code proxies may force internal browser structures into inconsistent states [31]. We provide examples of these problems in Section 2.4.

Figure 1(b) shows the architecture of the OP microkernel browser [25]. The core browser consists of a network stack, a storage system, and a user-interface system. Each component is isolated in a separate process, and they communicate with each other by exchanging messages through the kernel. A *web page instance* runs atop these core components. Each instance consists of an HTML parser/renderer, a JavaScript interpreter, an Xvnc [47] server, and zero or more plugins. All of these are isolated in separate processes and communicate via message passing. For example, the JavaScript interpreter sends messages to the HTML parser to



(a) A monolithic browser like Firefox.

(b) The OP browser.

(c) Atlantis.

Figure 1: Browser architectures. Rectangles represent strong isolation containers (either processes or C# `AppDomains`). Rounded rectangles represent modules within the same container. Solid borders indicate a lack of extensibility. Dotted borders indicate partial extensibility, and no border indicates complete extensibility.

dynamically update a page's content; the parser sends screen updates to the Xvnc server, which forwards them to the UI component using the VNC protocol [47]. The kernel determines which plugins to load by inspecting the MIME types of HTTP fetches (e.g., `application/x-shockwave-flash`). The kernel loads each plugin in a separate process, and the plugins use message passing to update the display or the page's HTML content. IBOS [46] is another microkernel browser that uses a philosophically similar isolation scheme.

OP and IBOS provide better security and fault isolation than monolithic browsers. However, OP and IBOS still use standard, off-the-shelf browser modules to provide the DOM tree, the JavaScript runtime, and so on. Thus, these browsers still present web developers with the frustrations that we describe next.

2.4 The Challenges of Web Development

Each browser provides its own implementation of the standard components. These implementation families are roughly compatible with each other, but each one has numerous quirks and bugs. Since a browser's components are weakly introspectable at best, developers are forced to use conditional code paths and ad-hoc best practices to get sophisticated web applications running across different browsers. In the remainder of this section, we provide concrete examples of these development challenges.

2.4.1 *Event Handling*

To react to user inputs like mouse clicks, applications attach event handlers to DOM nodes. When the user generates an event, the browser creates a new JavaScript event object and traces a path through the DOM tree, invoking any event handlers along that path. The official DOM specification defines a three-phase propagation model. Let the target node of the event be the DOM node whose GUI representation initiated the event; for example, the target for a mouse click might be a `<button>` element. In the *capturing* phase, the browser delivers the event to nodes along the path from the root `<html>` tag down to the `<button>` tag. In the *target* phase, the `<button>`'s event handlers are invoked. In the *bubbling* phase, the event follows the reverse path of the capturing phase, moving upwards towards the DOM tree root. At any point, the event can be canceled by an event handler. This stops the event from traversing the rest of the propagation path.

IE9 supports the capturing phase, but IE8 does not—on IE8, event propagation starts at the target. This makes some web applications difficult to write on IE8 but easy to write on IE9. For example, Mugshot [31] is a tool for logging and replaying web applications. On browsers that support the capturing phase, logging all GUI events is straightforward—Mugshot simply installs capturing event handlers at the top of the DOM tree. On IE8, Mugshot has to use an assortment of hacks. Mugshot cannot just define top-level bubbling handlers, since these handlers would miss events that were canceled earlier in the propagation process. Furthermore, whereas all events (are supposed to) have a capture phase, the DOM specification states that some events do not have a bubbling phase. IE8 adheres to this part of the specification, so bubbling-phase logging handlers would miss non-bubbling events as well. To avoid these problems, Mugshot uses DOM extensions [27]. Unfortunately, as we describe later, DOM extensions are also problematic.

Another severe problem with event handling is that browsers do not completely agree upon the set of events that should be supported. Furthermore, even "cross-browser" events may have different semantics on different browsers. For example:

- All popular browsers ostensibly support the `blur` event, which fires when the user shifts input focus away from a DOM node. Unfortunately, Chrome and Safari do not consistently fire the event for all types of DOM nodes. Opera sometimes generates multiple events for a single underlying blur [41].
- Browsers allow web pages to generate synthetic events. The browser is supposed to handle these fake events in the same way that it handles real ones. However, this does not happen in practice. For example, on

Firefox and IE, generating a fake `mouseup` event on a drop-down selection box will not cause an item to actually be selected. Similarly, generating synthetic keypress events on a text input will not actually cause the displayed text to change [31].
- IE fires an event when the user copies from or pastes to the clipboard. Other browsers do not support these events.

Abstraction libraries like jQuery [5] try to hide many of these differences, but some problems remain unfixable because they arise from browser functionality that is simply missing or fundamentally broken. Indeed, inconsistent cross-browser event policies can cause abstraction libraries themselves to behave differently on different browsers [28]. Furthermore, if a library's browser-sniffing algorithm has a bug, the library may execute the wrong conditional code paths for the actual browser being used [18].

2.4.2 Parsing bugs

Using the `document.write()` call, JavaScript code can insert new HTML tags into a document as that document is being parsed. This introduces race conditions into the parsing process, since the parser may or may not receive the new HTML tokens before it consumes the original token stream. Different browsers resolve this race in different ways [37].

Buggy parsers can also lead to security problems. Recent versions of Firefox, Chrome, Safari, and IE are vulnerable to a CSS parsing bug that allow an attacker to steal private data from authenticated web sessions [26]. Microsoft recently issued a patch for IE8 [32] which fixed a bug in the JSON [11] parser. In both of these cases, web developers who were aware of the security problems were reliant on browser vendors to implement fixes; the page developers themselves could not ship their pages with a patched execution environment.

2.4.3 Rendering bugs

All browsers have a variety of rendering quirks. For example, in Firefox 3.x and IE8 (but not prior versions of IE), negative values for the CSS `word-spacing` property are incorrectly coerced to zero, leading to problems with visual alignment [30]. Various hacks are needed to get IE to properly render semi-transparent elements [22]. Safari and Chrome can incorrectly calculate element dimensions that are specified as a percentage of the enclosing container's size [9]. Major browsers also define non-standard CSS attributes which are not universally implemented and whose use will cause web pages to render differently on different browsers [24].

2.4.4 JavaScript/DOM incompatibilities

JavaScript supports object inheritance using prototypes instead of classes [20]. A prototype object is an exemplar which defines properties for all objects using that prototype. Each object has exactly one prototype. Inheritance hierarchies are created by setting the prototype field for an object which itself is used as a prototype.

JavaScript is a dynamic language which permits runtime modification of prototype objects. In theory, this allows an application to arbitrarily extend the behavior of predefined objects. In practice, extending DOM objects is extremely brittle [27, 31] because the properties of DOM prototype objects are bound to opaque native code that is implemented in different ways on different browsers. Thus, wrapping old prototype properties, adding new ones, or deleting old ones may result in success on one browser but a runtime failure on another. Different browsers also define different prototype inheritance chains, meaning that modifying the same parent prototype on two browsers may lead to different property definitions for descendant prototypes. Although DOM extension is a powerful feature, it introduces so many corner cases that developers of the popular Prototype JavaScript framework [10] decided to abandon the technique for newer releases of the framework [27].

JavaScript allows applications to define getter and setter functions for object properties. These functions allow an application to intercept reads and writes to a property. Much like extending DOM prototypes, shimming the properties of native objects is difficult to do robustly, and it may disrupt the browser's event dispatch process [31]. Despite the danger of these techniques, they remain tantalizingly attractive because they allow a page to work around the deficiencies of a browser's DOM implementation.

In summary, it is easy to write a simple web page that looks the same in all browsers and has the same functionality in all browsers. Unfortunately, web pages of even moderate sophistication quickly encounter inconsistencies and bugs in browser runtimes. This observation motivates the Atlantis design, which we describe next.

3. ATLANTIS DESIGN

Figure 1(c) depicts Atlantis' architecture. At the bottom level is the switchboard process, the device server, and the storage manager; in aggregate, we refer to these components as the *master kernel*. The switchboard creates the isolation containers for web pages, and routes messages between these containers. The device server arbitrates access to non-essential peripheral devices like web cameras and microphones. The storage manager provides a key/value interface for persistent data.

The storage space is partitioned into a single public area and multiple, private, per-domain areas.[1] Any domain can read from or write to the public area, but the storage manager authenticates all requests for domain-private data. When the switchboard creates a new isolation container for domain X, it gives X an authentication token. It also sends a message to the storage manager that binds the token to X. Later, when X wishes to access private storage, it must include its authentication token in the request. In Section 3.2, we explain the usefulness of unauthenticated public storage.

In Atlantis, each instantiation of a web domain receives a separate isolation container. Following the terminology of Gazelle, we refer to these containers as principal instances. For example, if a user opens two separate tabs for the URL `http://a.com/foo.html`, Atlantis creates two separate principal instances. Each one contains a *per-instance Atlantis*

[1]A "domain" is synonymous with a `<protocol, host name, port>` origin as defined by the same-origin policy.

```
<environment>
  <compiler='http://a.com/compiler.syp'>
  <markupParser='http://b.com/parser.js'>
  <runtime='http://c.com/runtime.js'>
</environment>
```

Figure 2: To redefine its runtime, a web application puts an `<environment>` tag at the top of its markup.

kernel and a *script interpreter*. The instance kernel contains two modules. The network manager implements transfer protocols like `http://` and `file://`. The UI manager creates a new C# `Form` and registers handlers for low-level GUI events on that form. The UI manager forwards these events to the application-defined runtime, and updates the `Form`'s bitmap in response to messages from the page's layout engine.

The script interpreter executes abstract syntax trees which represent a new language called Syphon (§3.3). A web page installs a custom HTML/CSS parser, DOM tree, layout engine, and high-level script runtime by compiling the code to Syphon ASTs and submitting the ASTs for execution. The Syphon interpreter ensures that the code respects same-origin policies, but in all other regards, the interpreter is agnostic about what the code is doing. Thus, unlike in current browsers, the bulk of the web stack has no deep dependencies on internal browser state.

A principal instance's network stack, UI manager, and Syphon interpreter run in separate native threads. Thus, these components can run in parallel and take full advantage of multicore processors. Although these threads reside within a single process belonging to the principal instance, the threads are strongly isolated from each other using C# `AppDomains` [2]. `AppDomains` use the managed `.NET` runtime to enforce memory safety *within* a single process. Code inside an `AppDomain` cannot directly access memory outside its domain. However, domains can explicitly expose entry points that are accessible by other domains. The C# compiler translates these entry points into RPCs, serializing and deserializing data as necessary.

As shown in Figure 1(c), an application's runtime modules execute within the `AppDomain` of the interpreter. However, Syphon provides several language primitives which allow these modules to isolate themselves from each other. For example, an application can partition its Syphon code into privileged and unprivileged components, such that only privileged code can make kernel calls. We provide a detailed discussion of Syphon's protection features in Section 3.3. For now, we simply note that an application uses these isolation features to protect itself from itself—the Syphon interpreter is agnostic to the meaning of the protection domains that it enforces, and Atlantis' security guarantees do not depend on applications using Syphon-level protection mechanisms.

3.1 Initializing a New Principal Instance

When a new instance kernel starts, it receives a storage authentication token from the master kernel and then initializes its UI manager, network stack, and Syphon interpreter. Once this is done, the instance kernel fetches the markup associated with its page's URL. Atlantis is agnostic about whether this markup is HTML or something else. However, pages that wish to redefine their runtime must include a special `<environment>` tag at the top of their markup. An example of this tag is shown in Figure 2. The tag contains at most three elements.

- The `<compiler>` element specifies the code that will translate a page's script source into Syphon ASTs. The compiler itself must already be compiled to Syphon. If no compiler is specified, Atlantis assumes that the page's runtime environment is directly expressed in Syphon.
- The `<markupParser>` element specifies the code that the page will use to analyze its post-`<environment>` tag markup.
- The `<runtime>` code provides the rest of the execution environment, e.g., the layout engine, the DOM tree, and the high-level scripting runtime.

The compiler code must define a standardized entry point called `compiler.compile(srcString)`. This method takes a string of application-specific script code as input, and outputs the equivalent Syphon code. The instance kernel invokes `compiler.compile()` to generate executable code for the markup parser and the runtime library. After installing this code, the kernel passes the application's markup to the standardized parser entry point `markup.parse(markupStr)`. At this point, Atlantis relinquishes control to the application. As the application parses its markup, it invokes the kernel to fetch additional objects, update the screen, and so on.

If the instance kernel does not find an `<environment>` prefix in the page's markup, it assumes that the page wishes to execute atop the traditional web stack. In this case, Atlantis loads its own implementation of the HTML/CSS/JavaScript environment. From the page's perspective, this stack behaves like the traditional stack, with the important exception that everything is written in pure JavaScript, with no dependencies on shadow browser state. This means that, for example, modifying DOM prototypes will work as expected (§2.4), and placing getters or setters on DOM objects will not break event propagation. Of course, Atlantis' default web stack might have bugs, *but the application can fix these bugs itself without fear of breaking the browser.*

3.2 The Kernel Interface

As the web application executes, it interacts with its instance kernel using the API in Figure 3. The API is largely self-explanatory, but we highlight a few of the subtler aspects in the text below.

To create a new frame or tab, the application invokes `createPI()`. If the new principal instance is a child frame, the instance kernel in the parent registers the parent-child relationship with the master kernel. Later, if the user moves or resizes the window containing the parent frame, the master kernel notifies the instance kernels in the descendant frames, allowing Atlantis to maintain the visual relationships between parents and children.

`createPI(url, width, height, topX, topY, isFrame=false)`	Create a new principal instance. If `isFrame` is true, the new instance is the child of a parent frame. Otherwise, the new instance is placed in a new tab.
`registerGUICallback(dispatchFunc)`	Register an application-defined callback which the kernel will invoke when GUI events are generated.
`renderImage(pixelData, width, height, topX, topY, options)` `renderText(textStr, width, height, topX, topY, options)` `renderGUIwidget(widgetType, options)`	The application's layout engine uses these calls to update the screen. Strictly speaking, `renderImage()` is sufficient to implement a GUI. However, web pages that want to mimic the native look-and-feel of desktop applications can use native fonts and GUI widgets using `renderText()` and `renderWidget()`.
`HTTPStream openConnection(url)`	Open an HTTP connection to the given domain. Returns an object supporting blocking writes and both blocking and non-blocking reads.
`sendToFrame(targetFrameUrl, msg)`	Send a message to another frame. Used to implement cross-frame communication like `postMessage()`.
`executeSyphonCode(ASTsourceCode)`	Tell the interpreter to execute the given AST.
`persistentStore(mimeType, key, value, isPublic, token)` `string persistentFetch(mimeType, key, isPublic, token)`	Access methods for persistent storage. The storage volume is partitioned into a single public area, and multiple, private, per-domain areas. The `token` argument is the authentication nonce created by the switchboard.

Figure 3: Primary Atlantis kernel calls.

Using the `sendToFrame()` kernel call, two frames can exchange messages. An application can implement JavaScript's `postMessage()` as a trivial wrapper around `sendToFrame()`. The application can also use `sendToFrame()` to support cross-frame namespace abstractions. For example, in the traditional web stack, if a child frame and a parent frame are in the same domain, they can reference each other's JavaScript state through objects like `window.parent` and `window.frames[childId]`. An Atlantis DOM implementation supports these abstractions by interposing on accesses to these objects and silently generating `postMessage()` RPCs which read or write remote variables. We describe how Syphon supports such interpositioning in Section 3.3.

The functions `persistentStore()` and `persistentFetch()` allow applications to implement abstractions like cookies and DOM storage [48], a new HTML5 facility which provides a public storage area and private, per-domain storage. Atlantis' persistent store exports a simple key/value interface, and like DOM storage, it is split into a single public space and multiple, private, per-domain areas. Accessing private areas requires an authentication token; accessing public areas does not. The standard browser cache is stored in the public area, with cached items keyed by their URL. However, only instance kernels can write to public storage using URL keys. This ensures that when the network stack is handling a fetch for an object, it can trust any cached data that it finds for that object.

The public storage area is useful for implementing asynchronous cross-domain message queues. In particular, public storage allows two domains to communicate without forcing them to use `postMessage()` (which only works when both domains have simultaneously active frames). Atlantis does not enforce mandatory access controls for the public storage volume, so domains that desire integrity and authenticity for public data must layer security protocols atop Atlantis' raw storage substrate.

3.3 Syphon: Atlantis ASTs

Applications pass abstract syntax trees to Atlantis for execution. However, we could have chosen for applications to pass low-level bytecodes ala applets. We eschewed this option for two reasons. First, it is easier to optimize ASTs than bytecodes, since bytecodes obscure semantic relationships that must be recreated before optimizations can take place [45]. Second, it is difficult to reconstruct source code from bytecode, whereas this is trivial for ASTs. This feature is useful when one is debugging an application consisting of multiple scripts from multiple authors.

Atlantis ASTs encode a new language that we call *Syphon*. The Syphon specification is essentially a superset of the recent ECMAScript JavaScript specification [14], albeit described with a generic tree syntax that is amenable to serving as a compilation target for higher-level languages that may or may not resemble JavaScript. In this section, we focus on the Syphon features that ease the construction of robust, application-defined runtimes.

Object shimming: JavaScript supports getter and setter functions which allow applications to interpose on reads and writes to object properties. Syphon supports these, but it also introduces *watcher functions* which can execute when any property on an object is accessed in any way, including attempted deletions.

Watchers are very powerful, and the default Atlantis web stack uses them extensively. As we describe in Section 6.2, watchers allow the web stack to place input sanitizers in the write path of sensitive runtime variables that deal with untrusted user inputs. As another example, consider cross-frame namespace accesses like `window.parent.objInParent`. To implement this operation, the web stack defines a watcher on the `window.parent` object and resolves property accesses by issuing `sendToFrame()` calls to a namespace server in the parent frame.

Method binding and privileged execution: The typical Atlantis application will consist of low-level code like the layout engine and the scripting runtime, and higher-level code which has no need to directly invoke the Atlantis kernel. Syphon provides several language features that allow applications to isolate the low-level code from the high-level code, effectively creating an application-level kernel. Like in JavaScript, a Syphon method can be assigned to an arbitrary object and invoked as a method of that object. However, Syphon supports the binding of a method to a specific object, thereby preventing the method from being invoked with an arbitrary `this` reference.

Syphon also supports the notion of *privileged execution*. Syphon code can only invoke kernel calls if the code belongs to a privileged method that has a privileged `this` reference. By default, Syphon creates all objects and functions as privileged. However, an application can call Syphon's `disableDefaultPriv()` function to turn off that behavior. Subsequently, only privileged execution contexts will be able to create new privileged objects.

Our demonstration web stack leverages privileged execution, method binding, and watchers to prevent higher-level application code from arbitrarily invoking the kernel or perturbing critical data structures in the DOM tree. However, the Atlantis kernel is agnostic as to whether the application takes advantage of features like privileged execution. These features have no impact on Atlantis' security guarantees—they merely help an application to protect itself from itself.

Strong typing: By default, Syphon variables are untyped, as in JavaScript. However, Syphon allows programs to bind variables to types, facilitating optimizations in which the script engine generates fast, type-specific code instead of slow, dynamic-dispatch code for handling generic objects.

Note that strong primitive types have straightforward semantics, but the meaning of a strong object type is unclear in a dynamic, prototype-based language in which object properties, including prototype references, may be fluid. To better support strong object types, Syphon supports ECMAScript v5 notions of *object freezing*. By default, objects are `Unfrozen`, meaning that their property list can change in arbitrary ways at runtime. An object which is `PropertyListFrozen` cannot have old properties deleted or new ones added. A `FullFreeze` object has a frozen property list, and all of its properties are read-only. An object's freeze status can change dynamically, but only in the stricter direction. By combining strong primitive typing with object freezing along a prototype chain, Syphon can simulate traditional classes in strongly typed languages.

Threading: Syphon supports a full threading model with locks and signaling. Syphon threads are more powerful than HTML5 web workers [50] for two reasons. First, web workers cannot access native objects like the DOM tree because this would interfere with the browser's internal locking strategies. In contrast, Syphon DOM trees reside in application-layer code; this means that, for example, an application can define a multi-threaded layout engine without fear of corrupting internal browser state.

The second disadvantage of web workers is that they are limited to communication via asynchronous message exchanges. To implement these exchanges, browsers must serialize and deserialize objects across threading containers and fire notification callbacks. Syphon threads avoid this overhead since they are just thin wrappers around native OS threads.

3.4 Hardware Access

JavaScript has traditionally lacked access to hardware devices like web cameras and microphones. Thus, web pages that wished to access such devices were forced to use native code plugins such as Flash and Silverlight. Like Gazelle and OP, Atlantis loads plugins in separate processes and restricts their behavior using same-origin checks. We refer the reader to other work for a more detailed discussion of plugin isolation [25, 49].

The HTML5 specification [55] exposes hardware to JavaScript programs through a combination of new HTML tags and new JavaScript objects. For example, the `<device>` tag [54] can introduce a web camera object into the JavaScript namespace; the JavaScript interpreter translates reads and writes of that object's properties into hardware commands on the underlying device. Similarly, the `navigator.geolocation` object exposes location data gathered from GPS, wireless signal triangulation, or IP address geolocation [53].

HTML5 has been welcomed by web developers because it finally gives JavaScript first-class access to hardware devices. However, HTML5 is problematic from the security perspective because it entrusts hardware security to the JavaScript interpreter of an unsandboxed browser. Interpreters are complex, buggy pieces of code. For example, there have been several attacks on the JavaScript garbage collectors in Firefox, Safari, and IE [12]. Once an HTML5 JavaScript interpreter is subverted, an attacker has full access to all of the user's hardware devices.

In contrast to HTML5, Atlantis sandboxes the Syphon interpreter, preventing it from directly accessing hardware. Instead, web pages use the Gibraltar AJAX protocol [3] to access hardware. The master kernel contains a *device server* running in an isolated process. The device server is a small program which directly manipulates hardware using native code, and exports a web server interface on the `localhost` address. Principal instances that wish to access hardware send standard AJAX requests to the device server. For example, a page that wants to access a web camera might send an AJAX request to `http://localhost/webCam`, specifying various device commands in the HTTP headers of the request. Users authorize individual web domains to access individual hardware devices, and the device server authenticates each hardware request by looking at its `referrer` HTTP header. This header identifies the URL (and thus the domain) that issued the AJAX request.

In Atlantis, each principal instance runs its own copy of the Syphon interpreter in a separate `AppDomain`. Thus, even if a malicious web page compromises its interpreter, it cannot learn which other domains have hardware access unless those domains willingly respond to `postMessage()` requests for that information. Even if domains collude in this fashion, the instance kernel implements the networking stack, so web pages cannot fake the referrer fields in their hardware requests unless they also subvert the instance kernel.

A variety of subtle authentication issues remain. There are also low-level engineering questions, such as how to facilitate device discovery, and how to create efficient device protocols atop HTTP. We defer a full discussion of these issues to other work [3].

4. PROTOTYPE IMPLEMENTATION

In this section, we briefly describe how we implemented our Atlantis prototype. We also describe some of the performance challenges that we faced. Many of these challenges will be obviated in the next version of Atlantis, which will use the advanced SPUR `.NET` environment [4]. Compared to the default `.NET` environment, SPUR has a faster core runtime and a more powerful JIT compiler.

The trusted computing base: The core Atlantis system contains 8634 lines of C# code. 4900 lines belong to the Syphon interpreter, 358 belong to the master kernel, and the remainder belong to the instance kernel and the messaging library shared by various components. We implemented the full kernel interface described in Section 3.2, but we have not yet ported any plugins to Atlantis.

The Syphon interpreter: The interpreter implements all of the language features mentioned in Section 3.3. The interpreter represents each type of Syphon object as a subclass of the `SyphonObject` C# class. `SyphonObject` implements the "object as dictionary" abstraction used by all non-primitive types; it also implements introspection interfaces like watcher shims.

Dynamic languages like Syphon allow applications to manipulate the namespace in rich ways at runtime. For example, programs can dynamically add and remove variables from a scope; programs can also create closures, which are functions that remember the values of nonlocal variables in the enclosing scope. Each scope in the scope chain is essentially a dynamically modifiable dictionary, so each namespace operation requires the modification of one or more hash tables.

These modifications can be expensive, so the Syphon interpreter performs several optimizations to minimize dictionary operations. For example, when a function references a variable for the first time, the interpreter searches the scope chain, retrieves the relevant object, and places it in a unified cache that holds variables from various levels in the scope hierarchy. This prevents the interpreter from having to scan a potentially deep scope chain for every variable access.

Each function call requires the interpreter to allocate and initialize a new scope dictionary, and each function return requires a dictionary to be torn down. To avoid these costs, the interpreter tries to inline functions, twizzling the names of their arguments and local variables, rewriting the function code to reference these twizzled variables, and embedding these variables directly in the name cache of the caller. The function call itself is implemented as a direct branch to the rewritten function code. When the inlined function returns, the interpreter must destroy the twizzled name cache entries, but the overall cost of invoking an inlined function is still much smaller than the cost of invoking a non-inlined one. The interpreter will inline closures, but not functions that generate closures, since non-trivial bookkeeping is needed to properly bind closure variables.

The interpreter itself (and the enclosing browser) are written in C# and statically compiled to a CIL bytecode program. When the user invokes the browser, it is dynamically translated to x86 by the default `.NET` just-in-time compiler. In our current prototype, the Syphon interpreter compiles ASTs to high-level bytecodes, and then interprets those bytecodes directly. We did write another Syphon interpreter that directly compiled ASTs to CIL, but we encountered several challenges to making it fast. For example, to minimize the overhead of Syphon function calls, we wanted to implement them as direct branches to the starting CIL instructions of the relevant functions. Experiments showed that this kind of invocation was faster than placing the CIL for each Syphon function inside a C# function and then invoking that C# function using the standard CIL `CallVirt` instruction. Unfortunately, CIL does not support indirect branches. This made it tricky to implement function returns, since any given function can return to many different call sites. We had to implement function returns by storing call site program counters on a stack, and upon function return, using the topmost stack entry to index into a CIL switch where each case statement was a direct branch to a particular call site.

Unfortunately, this return technique does not work as described thus far. CIL is a stack-based bytecode, and the default `.NET` JIT compiler assumes that, for any instruction that is only the target of backward branches, the evaluation stack is empty immediately before that instruction executes [42]. This assumption was intended to simplify the JIT compiler, since it allows the compiler to determine the stack depth at any point in the program using a single pass through the CIL. Unfortunately, this assumption means that if a function return is a backwards branch to a call site, the CIL code after the call site must act as if the evaluation stack is initially empty; otherwise, the JIT compiler will declare the program invalid. If the evaluation stack is *not* empty before a function invocation (as is often the case), the application must manually save the stack entries to an application-defined data structure before branching to a function's first instruction.[2] Even with the overhead of manual stack management, branching function calls and returns were still faster than using the `CallVirt` instruction. However, this overhead did reduce the overall benefit of the technique, and the overhead would be avoidable with a JIT compiler that did not make the empty stack assumption.

We encountered several other challenges with the default JIT compiler. For example, we found that the JIT compiler was quick to translate the statically generated CIL for the Atlantis interpreter, but much slower to translate the dynamically generated CIL representing the Syphon application. For example, on several macrobenchmarks, we found that the CIL-emitting interpreter was spending twice as much time in JIT compilation as the high-level bytecode

[2]One could use forward branches to implement function returns, but then function *invocations* would have to use backwards branches, leading to a similar set of problems.

interpreter, even though the additional CIL to JIT (the CIL belonging to the Syphon program) was much smaller than the CIL for the interpreter itself.

Given these issues, our current prototype directly interprets the high-level bytecode. However, it is important to note that our challenges did not arise from an intrinsic deficiency in the `.NET` design, but from artifacts of the default `.NET` runtime, which is not optimized for our unusual demands. The SPUR project [4] has shown that significant performance gains can be realized by replacing the default `.NET` JIT engine with a custom one that can perform advanced optimizations like type speculation and trace-based JITing. We plan on using the SPUR framework in the next version of Atlantis.

The default web stack: If a web application does not provide its own high-level runtime, it will use Atlantis' default stack. This stack contains 5581 lines of JavaScript code which we compiled to Syphon ASTs using an ANTLR [40] tool chain. 65% of the code implements the standard DOM environment, providing a DOM tree, an event handling infrastructure, AJAX objects, and so on. The remainder of the code handles markup parsing, layout calculation, and rendering. Our DOM environment is quite mature, but our parsing and layout code is the target of active development; the latter set of modules are quite complex and require clever optimizations to run quickly.

5. EXOKERNEL BROWSERS: PRACTICAL ISSUES

Current web browsers must support an API that is hopelessly complex. This API is an uneasy conglomerate of disparate standards that define network protocols, markup formats, hardware interfaces, and more. Using exokernel principles [16], Atlantis allows each web page to ship with its own implementation of the web stack. Each page can tailor its execution environment to its specific needs; in doing so, the page liberates browser vendors from the futile task of creating a one-size-fits-all web stack.

Individual exokernel vendors might still produce buggy exokernel implementations. However, exokernel browsers are much simpler than their monolithic cousins; thus, their bugs should be smaller in number and easier to fix. Of course, an exokernel browser is only interesting when paired with a high-level runtime. In a certain sense, each high-level runtime represents yet another browser target with yet another set of quirks and incompatibilities that developers must account for. However, and importantly, *a page has complete control over its high-level runtime.* A page chooses which runtime it includes, and can modify that runtime as it sees fit. Thus, from the perspective of a web developer reasoning about portability challenges, the introduction of a new exokernel browser seems much less vexing than the introduction of a new monolithic browser.

Even if a single exokernel interface becomes the de facto browser design, there is always the danger that individual exokernel vendors will expand the narrow interface or introduce non-standard semantics for the sake of product differentiation. It seems impossible to prevent such feature creep by fiat. However, we believe that innovation at a low semantic level happens more slowly than innovation at a high semantic level. For example, fundamentally new file system features are created much less frequently than new application types that happen to leverage the file system. Thus, we expect that cross-vendor exokernel incompatibilities will arise much less frequently than incompatibilities between different monolithic browsers.

Exokernel browsers allow individual web applications to define their own HTML parsers, DOM trees, and so on. Multiple implementations of each component will undoubtedly arise. By design or accident, these implementations may become incompatible with each other. Furthermore, certain classes of components may be rendered unnecessary for some web applications; for example, if an application decides to use SGML instead of HTML as its markup language, it has no need for an HTML parser. Incompatibilities above the exokernel layer are not problematic, and are encouraged by Atlantis in the sense that Atlantis enables web developers to customize their high-level runtimes as they see fit. In practice, most Atlantis developers will not create runtimes from scratch, in the same way that most web developers today do not create their own JavaScript GUI frameworks. Instead, most Atlantis applications will use stock runtimes that are written by companies or open-source efforts, and which are incorporated into applications with little or no modification. Only popular sites or those with uncommon needs will possess the desire and the technical skill needed to write a heavily customized runtime.

6. EVALUATION

In this section, we explore three issues. First, we discuss the security of the Atlantis browser with respect to various threats. Second, we demonstrate how easy it is to extend the demonstration Atlantis web stack. Finally, we examine the performance of Atlantis on several microbenchmarks and macrobenchmarks. We defer an evaluation of Atlantis' AJAX hardware protocol to other work [3].

6.1 Security

Prior work has investigated the security properties of microkernel browsers [25, 46, 49]. Here, we briefly summarize these properties in the context of Atlantis, and explain why Atlantis provides stronger security guarantees than prior microkernel browsers.

Trusted computing base: The core Atlantis runtime contains 8634 lines of trusted C# code. This code belongs to the Syphon interpreter, the instance kernel, the master kernel, and the IPC library. In turn, these modules depend on the `.NET` runtime which implements the garbage collector, the standard `.NET` data types, and so on. The `.NET` runtime is also included in Atlantis' trusted computing base. However, these libraries are type-safe and memory managed, in contrast to the millions of lines of non-type safe C++ code found in IE, Firefox, and other commodity browsers. Thus, we believe that Atlantis' threat surface is comparatively much smaller, particularly given its narrow microkernel interface.

Our Atlantis prototype also includes 5581 lines of JavaScript representing the demonstration web stack, and an ANTLR [40] tool chain which compiles JavaScript to Syphon ASTs. These components are not part of the trusted computing base, since Atlantis does not rely on information from the high-level web stack to guide security decisions.

Principal Isolation: Like other microkernel browsers, Atlantis strongly isolates principal instances from each other and the core browser components. This prevents a large class of attacks which exploit the fact that monolithic browsers place data from multiple domains in the same address space, and lack a centralized point of enforcement for same-origin checks [6]. By strongly isolating each plugin in a process and subjecting them to the same-origin checks experienced by other web content, Atlantis prevents the full browser compromise that results when a monolithic browser has a subverted plugin in its address space [25, 49].

Gazelle, OP, and IBOS use a single browser kernel which, although memory isolated, is shared by all principal instances. If this kernel is compromised, the entire browser is compromised. For example, a subverted Gazelle kernel can inspect all messages exchanged between principal instances, tamper with persistent data belonging to an arbitrary domain, and update the visual display belonging to an arbitrary domain. In contrast, Atlantis has a single master kernel and multiple, sandboxed per-instance kernels. A subverted instance kernel can draw to its own rendering area and create new rendering areas, but it cannot access or update the display of another instance. Similarly, a subverted instance kernel can only tamper with public persistent data (which is untrustworthy by definition) or private persistent data that belongs to the domain of the compromised principal instance. To tamper with resources belonging to arbitrary domains, the attacker must subvert the master kernel, which is strongly isolated from the instance kernels.

Enforcing the Same-origin Policy: Browsers define an object's origin as the server hostname, port, and protocol which are used to fetch the object. For example, a script named `https://x.com:8080/y.js` has a protocol of `https`, a port of 8080, and a hostname of `x.com`.

The same-origin policy constrains how documents and scripts from domain `X` can interact with documents and scripts from domain `Y`. For example, JavaScript in `X`'s pages cannot issue `XMLHttpRequests` for `Y`'s JavaScript files; this ostensibly prevents `X`'s pages from reading `Y`'s source code. `X` can execute (but not inspect) `Y`'s code by dynamically creating a `<script>` tag and setting its `src` attribute to an object in `Y`'s domain. This succeeds because HTML tags are exempt from same-origin checks.

Importantly, the same-origin policy does not prevent colluding domains from communicating. For example, if `X` and `Y` have frames within the same page, the frames cannot forcibly inspect each other's cookies, nor can they forcibly read or write each other's DOM tree or JavaScript state. However, colluding domains can exchange arbitrary data across frames using `postMessage()`. Domains can also leak data through `iframe` URLs. For example, `X` can dynamically create an `iframe` with a URL like `http://y.com?=PRIVATE_X_DATA`. Given all of this, the practical implication of the same-origin policy is that it prevents *non-colluding* domains from tampering with each other.

As a result of Atlantis' exokernel design, it can perform many, but not all, of the origin checks that current browsers perform. However, Atlantis provides the same *practical* level of domain isolation. Each principal instance resides in a separate process, so each frame belonging to each origin is separated by hardware-enforced memory protection. This prevents domains from directly manipulating each other's JavaScript state or window properties. The kernel also partitions persistent storage by domain, ensuring that pages cannot inspect the cookies, DOM storage, or other private data belonging to external domains.

In Atlantis, abstractions like HTML tags and `XMLHttpRequest` objects are implemented entirely outside the kernel. Thus, when the Atlantis kernel services an `openConnection()` request, it cannot determine whether the fetch was initiated by an HTML parser upon encountering a `<script>` tag, or by an `XMLHttpRequest` fetch. To ensure that `<script>` fetches work, Atlantis must also allow cross-domain `XMLHttpRequest` fetches of JavaScript. This violates a strict interpretation of the same-origin policy, but it does not change the practical security provided by Atlantis, since `X`'s pages can trivially learn `Y`'s JavaScript source by downloading the `.js` files through `X`'s web server. From the security perspective, it is not important to prevent the discovery of inherently public source code. Instead, it is important to protect the *user-specific client-side state* which is exposed through the browser runtime and persistent client-side storage. Atlantis protects these resources using strong memory isolation and partitioned local storage. This is the same security model provided by Gazelle [49], which assumes that principal instances will not issue cross-domain script fetches for the purposes of inspecting source code.

6.2 Extensibility

Atlantis' primary goal is to allow web pages to customize their runtime in a robust manner. To demonstrate Atlantis' extensibility, we produced two variants of the default Atlantis web stack.

Safe `innerHTML`: A DOM node's `innerHTML` property is a text string representing the contents of its associated HTML tag. Web pages like message boards often update themselves by taking user-submitted text and assigning it to an `innerHTML` property. Unfortunately, an attacker can use this vector to insert malicious scripts into the page [38]. To prevent this attack, we place a setter shim (§3.3) on `innerHTML` that invokes the Caja sanitizer library [33]. Caja strips dangerous markup from the text, and the setter assigns the safe markup to `innerHTML`.

Stopping drive-by downloads: Assigning a URL to a frame's `window.location` property makes the frame navigate to a new site. If a frame loads a malicious third party script, the script can manipulate `window.location` to trick users into downloading malware [8]. To prevent this, we place a setter on `window.location` that either prevents all assignments, or only allows assignments if the target URL is in a whitelist.

Implementing these extensions on Atlantis was trivial since the default DOM environment is written in pure JavaScript. Neither of these application-defined extensions are possible

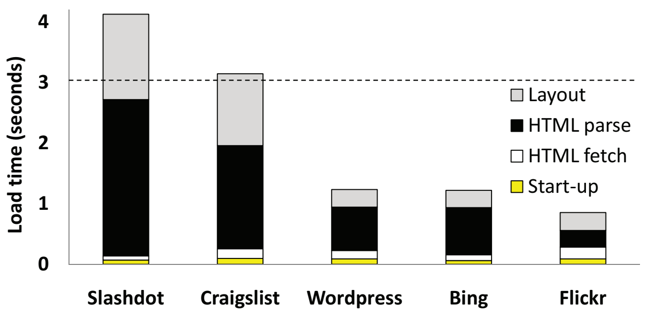


Figure 4: Atlantis page load times. The dotted line shows the three second window after which many users will become frustrated [21].

on a traditional browser. The design documents for popular browsers like IE and Firefox explicitly forbid applications from placing setters on `window` properties; placing setters on `innerHTML` is technically allowed, but actually doing so will break the browser's JavaScript engine [31].

6.3 Performance

To explore Atlantis' performance, we ran our prototype on a Lenovo Thinkpad laptop with 4 GB of RAM and a dual core 2.67 MHz processor. Our first experiment explored how quickly Atlantis could load several popular web pages. Figure 4 depicts the results. Each bar represents the average of five trials and contains four components: the start-up time between the user hitting "enter" on the address bar and the kernel issuing the fetch for the page's HTML; the time needed to fetch the HTML; the time needed to parse the HTML; and the time needed to calculate the layout and render the page. Note that layout time includes both pure computation time and the fetch delay for external content like images. To minimize the impact of network delays which Atlantis cannot control, Figure 4 depicts results for a warm browser cache. However, some objects were marked by their server as uncacheable and had to be refetched.

Given the unoptimized nature of our prototype scripting engine, we are encouraged by the results. In three of the five cases, Atlantis' load time is well below the three-second threshold at which users begin to get frustrated [21]. One page (Craiglist) is at the threshold, and another (Slashdot) is roughly a second over.

Figure 4 shows that Atlantis load times were often dominated by HTML parsing overhead. To better understand this phenomenon, we performed several microbenchmarks. The results are shown in Figure 5. Each bar represents Atlantis' relative execution speed with respect to IE8; standard deviations were less than 5% for each set of experiments. Figure 5 shows that in many cases, the Syphon interpreter is as fast or faster than IE's interpreter. In particular, Syphon is two to three times faster at accessing global variables, performing mathematical operations, and accessing object properties, whether they are defined directly on an object, on an object's prototype, or on a nested object four property accesses away.

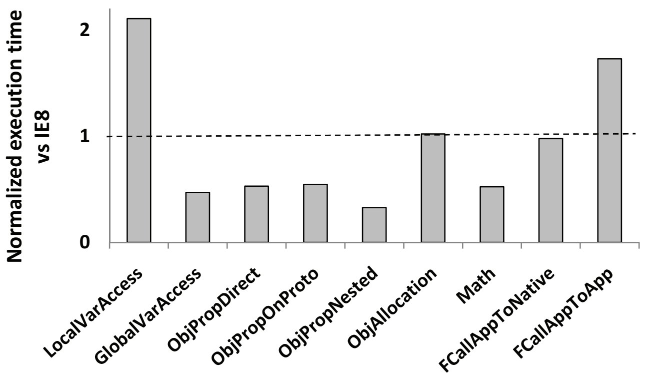


Figure 5: Execution speed versus IE8 (microbenchmarks).

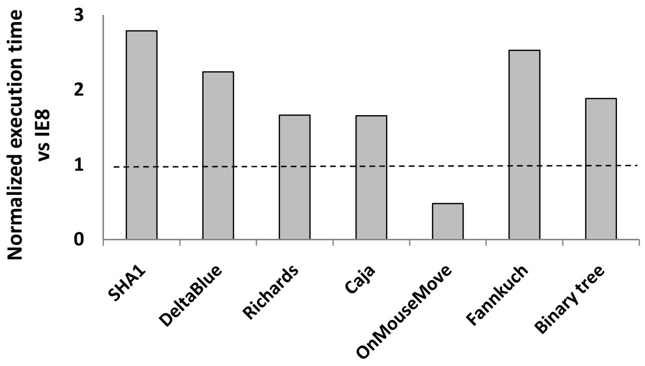


Figure 6: Execution speed versus IE8 (macrobenchmarks). All tests were CPU-bound except for `OnMouseMove`.

The cost of application code invoking native functions like `String.indexOf()` is the same on both platforms. However, Atlantis is twice as slow to access local variables, and 1.7 times as slow to invoke application-defined functions from other application-defined functions. Given that Atlantis accesses global variables much faster than IE, its relative slowness in accessing local variables is surprising—both types of accesses should be assisted by Atlantis' name cache. We are currently investigating this issue further. Atlantis function invocation is slower because Atlantis performs several safety checks that IE does not perform. These checks help to implement the Syphon language features described in Section 3.3. For example, to support strongly typed variables, the Syphon interpreter has to compare the type metadata for function parameters with the type metadata for the arguments that the caller actually supplied. To enforce privilege constraints, the Syphon interpreter must also check whether the function to invoke and its "this" pointer are both privileged. These checks make HTML parsing slow on our current Atlantis prototype, since the parsing process requires the invocation of many different functions that process strings, create new DOM nodes, and so on.

Figure 6 shows Atlantis' performance on several macrobenchmarks from three popular benchmark suites (SunSpider, Dromaeo, and Google's V8). Figure 6 shows that in general, IE8 is 1.5–2.8 times faster than our Atlantis prototype. However, for the `OnMouseMove` program, which tracks the rate at which the browser can fire application event handlers when the user rapidly moves the cursor, Atlantis is actually about 50% faster. This is important, since recent empirical work has shown that most web pages consist of a large number of small callback functions that are frequently invoked [43]. Note that firing application-defined event handlers requires native code to invoke application-defined code. The `FCallAppToNative` experiment in Figure 5 measures the costs for application code to call native code.

In summary, our prototype Atlantis implementation is already fast enough to load many web pages and to dispatch events at a fast rate. As mentioned in Section 4, we expect Atlantis' performance to greatly improve when we transition the code base from the stock `.NET` runtime to the SPUR [4] runtime which is aggressively tuned for performance.

7. RELATED WORK

In Sections 1.2, 2.3, and 6.1, we provide a detailed discussion of OP [25] and IBOS [46], two prior microkernel browsers. Gazelle [49] is another microkernel browser; like Atlantis, Gazelle is agnostic about a web page's runtime environment. However, Atlantis differs from Gazelle in five important ways.

- First, Gazelle only isolates web domains from each other; in contrast, Atlantis also protects intra-domain components like HTML parsers and scripting engines from each other.
- Second, Gazelle only has a single kernel, and uses heavy-weight processes to isolate this kernel from web page instances. In contrast, Atlantis uses lightweight C# `AppDomains` [2] to place a new kernel instance into each page's process. This maintains strong isolation between the kernel instance and the page while allowing Atlantis to sandbox individual kernel instances. Thus, unlike a Gazelle kernel subverted by domain `X`, a subverted Atlantis instance kernel can only tamper with data belonging to domain `X`.
- Third, Atlantis provides new, low-level runtime primitives (§3.3) which make it easier for web applications to define robust high-level runtimes.
- Fourth, Atlantis defines a bootstrapping process that allows a web page to automatically and dynamically load its own runtime environment. In contrast, Gazelle assumes that runtimes are manually installed using an out-of-band mechanism. Seamless dynamic loading is extremely important, since it allows page developers to change their runtime whenever they please without requiring action from the user.
- Finally, although the Gazelle kernel is agnostic to the web stack running above it, in practice, Gazelle uses an isolated version of Internet Explorer to provide that stack. In contrast, Atlantis provides a default web stack implemented in pure, extensible JavaScript. The stack includes a DOM tree, an HTML parser, a multi-pass layout algorithm, and a JavaScript runtime. This new stack provides web developers with unprecedented control over a runtime *implemented with their preferred software tools*: JavaScript and HTML.

The last point illuminates a primary contribution of this paper: whereas prior work leveraged microkernels solely to provide isolation, Atlantis leverages microkernels to also provide extensibility.

ServiceOS [34] is an extension of Gazelle that implements new policies for resource allocation. Architecturally, ServiceOS is very similar to Gazelle, so Atlantis has the same advantages over ServiceOS that it has over Gazelle.

JavaScript frameworks like jQuery [5] and Prototype [10] contain conditional code paths that try to hide browser incompatibilities. However, these libraries cannot hide all of these incompatibilities; furthermore, these libraries cannot make native code modules like layout engines amenable to introspection. Compile-to-JavaScript frameworks [7, 29] have similar limitations.

There are several JavaScript implementations of browser components like HTML parsers and JavaScript parsers [17, 23, 35, 44]. These libraries are typically used by a web page to analyze markup or script source before it is passed to the browser's actual parsing engine or JavaScript runtime. Using extensible web stacks, Atlantis lets pages extend and introspect the *real* application runtime. Atlantis' Syphon interpreter also provides new language primitives for making this introspection robust and efficient.

8. CONCLUSIONS

In this paper, we describe Atlantis, a new web browser which uses microkernels not just for security, but for extensibility as well. Whereas prior microkernel browsers reuse buggy, non-introspectable components from monolithic browsers, Atlantis allows each web page to define its own markup parser, layout engine, DOM tree, and scripting runtime. Atlantis gives pages the freedom to tailor their execution environments without fear of breaking fragile browser interfaces. Our evaluation demonstrates this extensibility, and shows that our Atlantis prototype is fast enough to render popular pages and rapidly dispatch event handlers. Atlantis also leverages multiple kernels to provide stronger security guarantees than previous microkernel browsers.

9. REFERENCES

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 2nd edition, 2007.

[2] J. Albahari and B. Albahari. *C# 3.0 in a Nutshell.* O'Reilly Publishing, O'Reilly Media, Inc., 3rd edition, 2007.

[3] Anonymous. Paper title blinded. In submission.

[4] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: A Trace-Based JIT Compiler for CIL. Microsoft Research Tech Report MSR-TR-2010-27, March 25, 2010.

[5] J. Chaffer and K. Swedberg. *jQuery 1.4 Reference Guide.* Packt Publishing, Birmingham, United Kingdom, 2010.

[6] S. Chen, D. Ross, and Y.-M. Wang. An Analysis of Browser Domain-Isolation Bugs and A Light-Weight Transparent Defense Mechanism. In *Proceedings of CCS*, Alexandria, VA, October 2007.

[7] R. Cooper and C. Collins. *GWT in Practice.* Manning Publications, Greenwich, CT, 2008.

[8] M. Cova, C. Kruegel, and G. Vigna. Detection and

Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *Proceedings of WWW*, Raleigh, NC, April 2010.

[9] C. Coyier. Percentage Bugs in WebKit. *CSS-tricks Blog*. http://css-tricks.com/percentage-bugs-in-webkit/, August 30, 2010.

[10] D. Crane, B. Bibeault, and T. Locke. *Prototype and Scriptaculous in Action*. Manning Publications, Greenwich, CT, 2007.

[11] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, July 2006.

[12] M. Daniel, J. Honoroff, and C. Miller. Engineering Heap Overflow Exploits with JavaScript. In *Proceedings of USENIX Workshop on Offensive Technologies*, 2008.

[13] J. Douceur, J. Elson, J. Howell, and J. Lorch. Leveraging Legacy Code to Deploy Desktop Applications on the Web. In *Proceedings of OSDI*, San Diego, CA, December 2008.

[14] Ecma International. Ecmascript language specification, 5^{th} edition, December 2009.

[15] H. Edskes. IE8 overflow and expanding box bugs. *Final Builds Blog*.http://www.edskes.net/ie/ie8overflowandexpandingboxbugs.htm, 2010.

[16] D. Engler, M. Kaashoek, and J. O. Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of SOSP*, Copper Mountain, CO, December 1995.

[17] Envjs Team. Envjs: Bringing the Browser. http://www.envjs.com/, 2010.

[18] eSpace Technologies. A tiny bug in Prototype JS leads to major incompatibility with Facebook JS client library. *eSpace.com blog*, April 23, 2008.

[19] Fielding, R., Gettys, J., Mogul, J.,Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

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

[21] Forrester Consulting. *eCommerce Web Site Performance Today: An Updated Look At Consumer Reaction To A Poor Online Shopping Experience*. White paper, 2009.

[22] S. Galineau. The CSS Corner: Using Filters In IE8. *IBBlog*. http://blogs.msdn.com/b/ie/archive/2009/02/19/the-css-corner-using-filters-in-ie8.aspx, February 19, 2009.

[23] D. Glazman. JSCSSP: A CSS parser in JavaScript. http://www.glazman.org/JSCSSP/, 2010.

[24] Google. Fixing Google Chrome Compatibility bugs in WebSites. http://code.google.com/p/doctype/wiki/ArticleGoogleChromeCompatFAQ#Inline_elements_can%27t_enclose_block_elements, May 25, 2010.

[25] C. Grier, , S. Tang, and S. King. Secure Web Browsing with the OP Web Browser. In *Proceedings of IEEE Security*, Oakland, CA, May 2008.

[26] L.-S. Huang, Z. Weinberg, C. Evans, and C. Jackson. Protecting Browsers from Cross-Origin CSS Attacks. In *Proceedings of CCS*, Chicago, IL, October 2010.

[27] J. Zaytsev. What's wrong with extending the DOM. *Perfection Kills Website*. http://perfectionkills.com/whats-wrong-with-extending-the-dom, April 5, 2010.

[28] jQuery Message Forum. Focus() inside a blur() handler. https://forum.jquery.com/topic/focus-inside-a-blur-handler, January 2010.

[29] N. Kothari. Script#: Version 0.5.5.0. http://projects.nikhilk.net/ScriptSharp, 2009.

[30] L. Lazaris. CSS Bugs and Inconsistencies in Firefox 3.x. *Webdesigner Depot*. http://www.webdesignerdepot.com/2010/03/css-bugs-and-inconsistencies-in-firefox-3-x, March 15, 2010.

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

[32] Microsoft. Update for Native JSON feature in IE8. http://support.microsoft.com/kb/976662, February 2010.

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

[34] A. Moshchuk and H. J. Wang. Resource Management for Web Applications in ServiceOS. Microsoft Research Tech Report MSR-TR-2010-56, May 18, 2010.

[35] Mozilla Corporation. Narcissus javascript. http://mxr.mozilla.org/mozilla/source/js/narcissus/.

[36] Mozilla Developer Center. Gecko Plugin API Reference. https://developer.mozilla.org/en/Gecko_Plugin_API_Reference, 2010.

[37] Mozilla Developer Center. HTML5 Parser. https://developer.mozilla.org/en/HTML/HTML5/HTML5_Parser, July 29, 2010.

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

[39] T. Olsson. *The Ultimate CSS Reference*. Sitepoint, Collingwood, Victoria, Austraiia, 2008.

[40] T. Parr. *The Definitive ANTLR Reference*. Pragmatic Bookshelf, Raleigh, North Carolina, 2007.

[41] Peter-Paul Koch. QuirksMode–for all your browser quirks. http://www.quirksmode.org, 2011.

[42] J. Pobar, T. Neward, D. Stutz, and G. Shilling. Shared Source CLI 2.0 Internals. http://callvirt.net/blog/files/Shared%20Source%20CLI%202.0%20Internals.pdf, 2008.

[43] P. Ratanaworabhan, B. Livshits, and B. Zorn. JSMeter: Comparing the Behavior of JavaScript Benchmarks with RealWeb Applications. In *Proceedings of USENIX WebApps*, Boston, MA, June 2010.

[44] J. Resig. Pure JavaScript HTML Parser. http://ejohn.org/blog/pure-javascript-html-parser/, May 2008.

[45] C. Stork, P. Housel, V. Haldar, N. Dalton, and M. Franz. Towards language-agnostic mobile code. In *Proceedings of the Workshop on Multi-Language Infrastructure and Interoperability*, Firenze, Italy, 2001.

[46] S. Tang, H. Mai, and S. T. King. Trust and Protection in the Illinois Browser Operating System. In *Proceedings of OSDI*, 2010.

[47] C. Tyler. *X Power Tools*. O'Reilly Media, Inc., Cambridge, MA, 2007.

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

[49] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-principal OS Construction of the Gazelle Web Browser. In *Proceedings of USENIX Security*, 2009.

[50] Web Hypertext Application Technology Working Group (WHATWG). Web Workers (Draft Recommendation). http://www.whatwg.org/specs/web-workers/current-work/, September 10, 2010.

[51] World Wide Web Consortium. Document object model (DOM) level 2 core specification. W3C

Recommendation, November 13, 2000.

[52] World Wide Web Consortium. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. W3C Working Draft. `http://www.w3.org/TR/CSS2`, September 8, 2009.

[53] World Wide Web Consortium. Geolocation API Specification. `http://dev.w3.org/geo/api/spec-source.html`, February 10, 2010.

[54] World Wide Web Consortium. HTML Device: An addition to HTML. `http://dev.w3.org/html5/html-device/`, September 9, 2010.

[55] World Wide Web Consortium. HTML5: A vocabulary and associated APIs for HTML and XHTML. W3C Working Draft. `http://www.w3.org/TR/html5`, June 24, 2010.

[56] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of IEEE Security*, Oakland, CA, May 2009.