C3: An Experimental, Extensible, Reconfigurable Platform for HTML-based Applications

Benjamin S. Lerner Brian Burg
University of Washington

Herman Venter Wolfram Schulte
Microsoft Research

Abstract

The common conception of a (client-side) *web application* is some collection of HTML, CSS and JavaScript (JS) that is hosted within a web browser and that interacts with the user in some non-trivial ways. The common conception of a *web browser* is a monolithic program that can render HTML, execute JS, and gives the user a portal to navigate the web. Both of these are misconceptions: nothing inherently confines webapps to a browser's page-navigation idiom, and browsers can do far more than merely render content. Indeed, browsers and web apps are converging in functionality, but their underlying technologies are so far largely distinct.

We present C3, an implementation of the HTML/CSS/JS platform designed for web-client research and experimentation. C3's typesafe, modular architecture lowers the barrier to webapp and browser research. Additionally, C3 explores the role of *extensibility* throughout the web platform for customization and research efforts, by introducing novel extension points and generalizing existing ones. We discuss and evaluate C3's design choices for flexibility, and provide examples of various extensions that we and others have built.

1 Introduction

We spend vast amounts of time using web browsers: casual users view them as portals to the web, while power users enjoy them as flexible, sophisticated tools with countless uses. Researchers of all stripes view browsers and the web itself as systems worthy of study: browsers are a common thread for web-related research in fields such as HCI, security, information retrieval, sociology, software engineering, and systems research. Yet today's production-quality browsers are all monolithic, complex systems that do not lend themselves to easy experimentation. Instead, researchers often must modify the source code of the browsers—usually tightly-optimized, obscure, and sprawling C/C++ code—and this requirement of deep domain knowledge poses a high barrier to entry, correctness, and adoption of research results.

Of course, this is a simplified depiction: browsers are not entirely monolithic. Modern web browsers, such as Internet Explorer, Firefox or Chrome, support *extensions*, pieces of code—usually a mix of HTML, CSS and JavaScript (JS)—that are written by third-party developers and downloaded by end users, that build on top of the browser and customize it dynamically at runtime.[1] To date, such customizations focus primarily on modifying the user interfaces of browsers. (Browsers also support *plug-ins*, binary components that provide functionality, such as playing new multimedia file types, not otherwise available from the base browser. Unlike extensions, plug-ins cannot interact directly with each other or extend each other further.)

Extensions are widely popular among both users and developers: millions of Firefox users have downloaded thousands of different extensions over two billion times[2]. Some research projects have used extensions to implement their ideas. But because current extension mechanisms have limited power and flexibility, many research projects still must resort to patching browser sources:

1. XML3D [14] defines new HTML tags and renders them with a 3D ray-tracing engine—but neither HTML nor the layout algorithm are extensible.

2. Maverick [12] permits writing device drivers in JS and connecting the devices (e.g., webcams, USB thumb drives, GPUs, etc.) to web pages—but JS cannot send raw USB packets to the USB root hub.

3. RePriv [5] experiments with new ways to securely expose and interact with private browsing informa-

[1] Opera supports widgets, which do not interact with the browser or content, and Safari recently added small but slowly-growing support for extensions in a manner similar to Chrome. We ignore these browsers in the following discussions.

[2] `https://addons.mozilla.org/en-US/statistics/`

tion (e.g. topics inferred from browsing history) via reference-monitored APIs—but neither plug-ins nor JS extensions can guarantee the integrity or security of the mined data as it flows through the browser.

These projects incur development and maintenance costs well above the inherent complexity of their added functionality. Moreover, patching browser sources makes it difficult to update the projects for new versions of the browsers. This overhead obscures the fact that such research projects are essentially extensions to the web-browsing experience, and would be much simpler to realize on a flexible platform with more powerful extension mechanisms. Though existing extension points in mainstream browsers vary widely in both design and power, none can support the research projects described above.

1.1 The extensible future of web browsers

Web browsers have evolved from their beginnings as mere document viewers into web-application runtime platforms. Applications such as Outlook Web Access or Google Documents are sophisticated programs written in HTML, CSS and JS that use the browser *only for rendering and execution* and ignore everything else browsers provide (bookmarks, navigation, tab management, etc.). Projects like Mozilla Prism[3] strip away all the browser "chrome" while reusing the underlying HTML/CSS/JS implementation (in this case, Gecko), letting webapps run like native apps, outside of the typical browser. Taken to an extreme, "traditional" applications such as Firefox or Thunderbird are written using Gecko's HTML/CSS/JS engine, and clearly are not themselves hosted within a browser.

While browsers and web apps are growing closer, they are still mostly separate with no possibility of tight, customizable integration between them. Blogging clients such as WordPress, instant messaging clients such as Gchat, and collaborative document editors such as Mozilla Skywriter are three disjoint web applications, all designed to create and share content. An author might be using all three simultaneously, and searching for relevant web resources to include as she writes. Yet the only way to do so is to "escape the system", copying and pasting web content via the operating system.

1.2 Contributions

The time has come to reconsider browser architectures with a focus on extensibility. We present C3: a reconfigurable, extensible implementation of HTML, CSS and JS designed for web client research and experimentation. C3 is written entirely in C# and takes advantage of .Net's libraries and type-safety. Similar to Firefox building atop Gecko, we have built a prototype browser atop C3, using only HTML, CSS and JS.

By *reconfigurable*, we mean that each of the modules in our browser—Document Object Model (DOM) implementation, HTML parser, JS engine, etc.—is loosely coupled by narrow, typesafe interfaces and can be replaced with alternate implementations compiled separately from C3 itself. By *extensible*, we mean that the default implementations of the modules support run-time extensions that can be systematically introduced to

1. extend the syntax and implementation of HTML
2. transform the DOM when being parsed from HTML
3. extend the UI of the running browser
4. extend the environment for executing JS, and
5. transform and modify running JS code.

Compared to existing browsers, C3 introduces novel extension points (1) and (5), and generalizes existing extension points (2)–(4). These extension points are treated in order in Section 3. We discuss their functionality and their security implications with respect to the same-origin policy [13]. We also provide examples of various extensions that we and others have built.

The rest of the paper is structured as follows. Section 2 gives an overview of C3's architecture and highlights the software engineering choices made to further our modularity and extensibility design goals. Section 3 presents the design rationale for our extension points and discusses their implementation. Section 4 evaluates the performance, expressiveness, and security implications of our extension points. Section 5 describes future work. Section 6 concludes.

2 C3 architecture and design choices

As a research platform, C3's explicit design goals are architectural modularity and flexibility where possible, instead of raw performance. Supporting the various extension mechanisms above requires hooks at many levels of the system. These goals are realized through careful design and implementation choices. Since many requirements of an HTML platform are standardized, aspects of our architecture are necessarily similar to other HTML implementations. C3 lacks some of the features present in mature implementations, but contains all of the essential architectural details of an HTML platform.

C3's clean-slate implementation presented an opportunity to leverage modern software engineering tools and practices. Using a managed language such as C# sidesteps the headaches of memory management, buffer overruns, and many of the common vulnerabilities in production

[3] http://prism.mozillalabs.com/

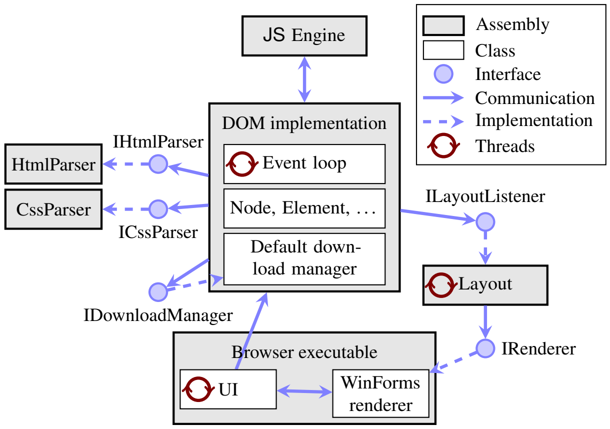

Figure 1: C3's modular architecture

browsers. Using a higher-level language better preserves abstractions and simplifies many implementation details. Code Contracts [4] are used throughout C3 to ensure implementation-level invariants and safety properties—something that is not feasible in existing browsers.

Below, we sketch C3's module-level architecture, and elaborate on several core design choices and resulting customization opportunities. We also highlight features that enable the extension points examined in Section 3.

2.1 Pieces of an HTML platform

The primary task of any web platform is to parse, render, and display an HTML document. For interactivity, web applications additionally require the managing of events such as user input, network connections, and script evaluation. Many of these sub-tasks are independent; Figure 1 shows C3's module-level decomposition of these tasks. The *HTML parser* converts a text stream into an object tree, while the *CSS parser* recognizes stylesheets. The *JS engine* dispatches and executes event handlers. The *DOM implementation* implements the API of DOM nodes, and implements bindings to expose these methods to JS scripts. The *download manager* handles actual network communication and interactions with any on-disk cache. The *layout engine* computes the visual structure and appearance of a DOM tree given current CSS styles. The *renderer* displays a computed layout. The browser's UI displays the output of the renderer on the screen, and routes user input to the DOM.

2.2 Modularity

Unlike many modern browsers, C3's design embraces loose coupling between browser components. For example, it is trivial to replace the HTML parser, renderer frontend, or JS engine without modifying the DOM implementation or layout algorithm. To make such drop-in replacements feasible, C3 shares *no* data structures between modules when possible (i.e., each module is heap-disjoint). This design decision also simplifies threading disciplines, and is further discussed in Section 2.7.

Simple implementation-agnostic interfaces describe the operations of the DOM implementation, HTML parser, CSS parser, JS engine, layout engine, and front-end renderer modules. Each module is implemented as a separate .Net assembly, which prevents modules from breaking abstractions and makes swapping implementations simple. Parsers could be replaced with parallel [8] or speculative[4] versions; layout might be replaced with a parallel [11] or incrementalizing version, and so on. The default module implementations are intended as straightforward, unoptimized reference implementations. This permits easy per-module evaluations of alternate implementation choices.

2.3 DOM implementation

The DOM API is a large set of interfaces, methods and properties for interacting with a document tree. We highlight two key design choices in our implementation: what the object graph for the tree looks like, and the bindings of these interfaces to C$^\#$ classes. Our choices aim to minimize overhead and "boilerplate" coding burdens for extension authors.

Object trees: The DOM APIs are used throughout the browser: by the HTML parser (Section 2.4) to construct the document tree, by JS scripts to manipulate that tree's structure and query its properties, and by the layout engine to traverse and render the tree efficiently. These clients use distinct but overlapping subsets of the APIs, which means they must be exposed both to JS and to C$^\#$, which in turn leads to the first design choice.

One natural choice is to maintain a tree of "implementation" objects in the C$^\#$ heap separate from a set of "wrapper" objects in the JS heap[5] containing pointers to their C$^\#$ counterparts: the JS objects are a "view" of the underlying C$^\#$ "model". The JS objects contain stubs for all the DOM APIs, while the C$^\#$ objects contain implementations and additional helper routines. This design incurs the overheads of extra pointer dereferences (from the JS APIs to the C$^\#$ helpers) and of keeping the wrappers synchronized with the implementation tree. However, it permits specializing both representations for their respective usages, and the extra indirection enables

[4] `http://hsivonen.iki.fi/speculative-html5-parsing/`

[5] Expert readers will recognize that "objects in the JS heap" are implemented by C$^\#$ "backing" objects; we are distinguishing these from C$^\#$ objects that do not "back" any JS object.

multiple views of the model: This is essentially the technical basis of Chrome extensions' "isolated worlds" [1], where the indirection is used to ensure security properties about extensions' JS access to the DOM. Firefox also uses the split to improve JS memory locality with "compartments" [15].

By contrast, C3 instead uses a single tree of objects visible to both languages, with each DOM node being a C# subclass of an ordinary JS object, and each DOM API being a standard C# method that is exposed to JS. This design choice avoids both overheads mentioned above. Further, Spur [3], the tracing JS engine currently used by C3, can trace from JS into DOM code for better optimization opportunities. To date, no other DOM implementation/JS engine pair can support this optimization.

DOM language bindings: The second design choice stems from our choice for the first: how to represent DOM objects such that their properties are callable from both C# and JS. This representation must be open: extensions such as XML3D must be able to define new types of DOM nodes that are instantiable from the parser (see Section 3.1) and capable of supplying new DOM APIs to both languages as well. Therefore any new DOM classes must subclass our C# DOM class hierarchy easily, and be able to use the same mechanisms as the built-in DOM classes. Our chosen approach is a thin marshaling layer around a C# implementation, as follows:

- All Spur JS objects are instances of C# classes deriving from `ObjectInstance`. Our DOM class hierarchy derives from this too, and so DOM objects *are* JS objects, as above.
- All JS objects are essentially property bags, or key/value dictionaries, and "native" objects (e.g. `Math`, `Date`) may contain properties that are implemented by the JS runtime and have access to runtime-internal state. All DOM objects are native, and their properties (the DOM APIs) access the internal representation of the document.
- The JS dictionary is represented within Spur as a `TypeObject` field of each `ObjectInstance`. To expose a native method on a JS object, the implementation simply adds a property to the `TypeObject` mapping the (JS) name to the (C#) function that implements it.[6] This means that a single C# function can be called from both languages, and need not be implemented twice.

The `ObjectInstance` and `TypeObject` classes are public Spur APIs, and so our DOM implementation is readily extensible by new node types.

2.4 The HTML parser

The HTML parser is concerned with transforming HTML source into a DOM tree, just as a standard compiler's parser turns source into an AST. Extensible compilers' parsers can recognize supersets of their original language via extensions; similarly, C3's default HTML parser supports extensions that add new HTML tags (which are implemented by new C# DOM classes as described above; see also Section 3.1).

An extensible HTML parser has only two dependencies: a means for constructing a new node given a tag name, and a factory method for creating a new node and inserting it into a tree. This interface is far simpler than that of any DOM node, and so exists as the separate `INode` interface. The parser has no hard dependency on a specific DOM implementation, and a minimal implementation of the `INode` interface can be used to test the parser independently of the DOM implementation. The default parser implementation is given a DOM node factory that can construct `INode`s for the built-in HTML tag names. Extending the parser via this factory is discussed in Section 3.1.

2.5 Computing visual structure

The layout engine takes a document and its stylesheets, and produces as output a *layout tree*, an intermediate data structure that contains sufficient information to display a visual representation of the document. The *renderer* then consults the layout tree to draw the document in a platform- or toolkit-specific manner.

Computing a layout tree requires three steps: first, DOM nodes are attributed with style information according to any present stylesheets; second, the layout tree's structure is determined; and third, nodes of the layout tree are annotated with concrete styles (placement and sizing, fonts and colors, etc.) for the renderer to use. Each of these steps admits a naïve reference implementation, but both more efficient and more extensible algorithms are possible. We focus on the former here; layout extensibility is revisited in Section 3.3.

Assigning node styles The algorithm that decorates DOM nodes with CSS styles does not depend on any other parts of layout computation. Despite the top-down implementation suggested by the name "cascading style sheets", several efficient strategies exist, including recent and ongoing research in parallel approaches [11].

Our default style "cascading" algorithm is self-contained, single-threaded and straightforward. It decorates each DOM node with an immutable calculated style object, which is then passed to the related layout tree

[6] Technically, to a C# function that unwraps the JS values into strongly-typed C# values, then calls a second C# function with them.

node during construction. This immutable style suffices thereafter in determining visual appearance.

Determining layout tree structure The layout tree is generated from the DOM tree in a single traversal. The two trees are approximately the same shape; the layout tree may omit nodes for invisible DOM elements (e.g. ⟨**script**/⟩), and may insert "synthetic" nodes to simplify later layout invariants. For consistency, this transformation must be serialized between DOM mutations, and so runs on the DOM thread (see Section 2.7). The layout tree must preserve a mapping between DOM elements and the layout nodes they engender, so that mouse movement (which occurs in the renderer's world of screen pixels and layout tree nodes) can be routed to the correct target node (i.e. a DOM element). A naïve pointer-based solution runs afoul of an important design decision: C3's architectural goals of modularity require that the layout and DOM trees share no pointers. Instead, all DOM nodes are given unique numeric ids, which are *preserved* by the DOM-to-layout tree transformation. Mouse targeting can now be defined in terms of these ids while preserving pointer-isolation of the DOM from layout.

Solving layout constraints The essence of any layout algorithm is to solve constraints governing the placement and appearance of document elements. In HTML, these constraints are irregular and informally specified (if at all). Consequently the constraints are typically solved by a manual, multi-pass algorithm over the layout tree, rather than a generic constraint-solver [11]. The manual algorithms found in production HTML platforms are often tightly optimized to eliminate some passes for efficiency.

C3's architecture admits such optimized approaches, too; our reference implementation keeps the steps separate for clarity and ease of experimentation. Indeed, because the layout tree interface does not assume a particular implementation strategy, several layout algorithm variants have been explored in C3 with minimal modifications to the layout algorithm or components dependent on the computed layout tree.

2.6 Accommodating Privileged UI

Both Firefox and Chrome implement some (or all) of their user interface (e.g. address bar, tabs, etc.) in declarative markup, rather than hard-coded native controls. In both cases this gives the browsers increased flexibility; it also enables Firefox's extension ecosystem. The markup used by these browsers is *trusted*, and can access internal APIs not available to web content. To distinguish the two, trusted UI files are accessed via a different URL scheme: e.g., Firefox's main UI is loaded using `chrome://browser/content/browser.xul`.

We chose to implement our prototype browser's UI in HTML for two reasons. First, we wanted to experiment with writing sophisticated applications entirely within the HTML/CSS/JS platform and experience first-hand what challenges arose. Even in our prototype, such experience led to the two security-related changes described below. Secondly, having our UI in HTML opens the door to the extensions described in Section 3; the entirety of a C3-based application is available for extension. Like Firefox, our browser's UI is available at a privileged URL: launching C3 with a command-line argument of `chrome://browser/tabbrowser.html` will display the browser UI. Launching it with the URL of any website will display that site without any surrounding browser chrome. Currently, we only permit HTML file resources bundled within the C3 assembly itself to be given privileged `chrome://` URLs.

Designing this prototype exposed deliberate limitations in HTML when examining the navigation history of child windows (popups or ⟨**iframe**/⟩s): the APIs restrict access to same-origin sites only, and are write-only. A parent window cannot see what site a child is on unless it is from the same origin as the parent, and can never see what sites a child has visited. A browser must avoid both of these restrictions so that it can implement the address bar.

Rather than change API visibility, C3 extends the DOM API in two ways. First, it gives privileged pages (i.e., from `chrome://` URLs) a new `childnavigated` notification when their children are navigated, just before the `onbeforeunload` events that the children already receive. Second, it treats `chrome://` URLs as trusted origins that *always pass same-origin checks*. The trusted-origin mechanism and the custom navigation event suffice to implement our browser UI.

2.7 Threading architecture

One important point of flexibility is the mapping between threads and the HTML platform components described above. We do not impose any threading discipline beyond necessary serialization required by HTML and DOM standards. This is made possible by our decision to prevent data races by design: in our architecture, data is either immutable, or it is not shared amongst multiple components. Thus, it is possible to choose any threading discipline within a single component; a single thread could be shared among all components for debugging, or several threads could be used within each component to implement worker queues.

Below, we describe the default allocation of threads among components, as well as key concurrency concerns for each component.

2.7.1 The DOM/JS thread(s)

The DOM event dispatch loop and JS execution are single-threaded within a set of related web pages[7]. "Separate" pages that are unrelated[8] can run entirely parallel with each other. Thus, sessions with several tabs or windows open simultaneously use multiple DOM event dispatch loops.

In C3, each distinct event loop consists of two threads: a *mutator* to run script and a *watchdog* to abort run-away scripts. Our system maintains the invariant that all mutator threads are *heap-disjoint*: JS code executing in a task on one event loop can only access the DOM nodes of documents sharing that event loop. This invariant, combined with the single-threaded execution model of JS (from the script's point of view), means all DOM nodes and synchronous DOM operations can be lock-free. (Operations involving local storage are asynchronous and must be protected by the storage mutex.) When a window or ⟨**iframe**/⟩ is navigated, the relevant event loop may change. An event loop manager is responsible for maintaining the mappings between windows and event loops to preserve the disjoint-heap invariant.

Every DOM manipulation (node creation, deletion, insertion or removal; attribute creation or modification; etc.) notifies any registered *DOM listener* via a straightforward interface. One such listener is used to inform the layout engine of all document manipulations; others could be used for testing or various diagnostic purposes.

2.7.2 The layout thread(s)

Each top-level browser window is assigned a *layout* thread, responsible for resolving layout constraints as described in Section 2.5. Several browser windows might be simultaneously visible on screen, so their layout computations must proceed in parallel for each window to quickly reflect mutations to the underlying documents. Once the DOM thread computes a layout tree, it transfers ownership of the tree to the layout thread, and begins building a new tree. Any external resources necessary for layout or display (such as image data), are also passed to the layout thread as uninterpreted .Net streams. This isolates the DOM thread from any computational errors on the layout threads.

2.7.3 The UI thread

It is common for GUI toolkits to impose threading restrictions, such as only accessing UI widgets from their creating thread. These restrictions influence the platform insofar as *replaced elements* (such as buttons or text boxes) are implemented by toolkit widgets.

C3 is agnostic in choosing a particular toolkit, but rather exposes abstract interfaces for the few widget properties actually needed by layout. Our prototype currently uses the .Net WinForms toolkit, which designates one thread as the "UI thread", to which all input events are dispatched and on which all widgets must be accessed. When the DOM encounters a replaced element, an actual WinForms widget must be constructed so that layout can in turn set style properties on that widget. This requires synchronous calls from the DOM and layout threads to the UI thread. Note, however, that *responding* to events (such as mouse clicks or key presses) is *asynchronous*, due to the indirection introduced by numeric node ids: the UI thread simply adds a message to the DOM event loop with the relevant ids; the DOM thread will process that message in due course.

3 C3 Extension points

The extension mechanisms we introduce into C3 stem from a principled examination of the various semantics of HTML. Our interactions with webapps tacitly rely on manipulating HTML in two distinct ways: we can interpret it *operationally* via the DOM and JS programs, and we can interpret it *visually* via CSS and its associated layout algorithms. Teasing these interpretations apart leads to the following two transformation pipelines:

- $\text{JS global object + HTML source}^{1,2}$

  $\xrightarrow{\text{HTML } parsing} {}^{3}\text{DOM subtrees}^{4}$

  $\xrightarrow{\texttt{onload}} \text{DOM document}^{5}$

  $\xrightarrow{\text{JS } events} \text{DOM document}\ldots$

- $\text{DOM document + CSS source}^{6}$

  $\xrightarrow{\text{CSS } parsing} \text{CSS content model}^{7}$

  $\xrightarrow{layout} \text{CSS box model}$

The first pipeline distinguishes four phases of the document lifecycle, from textual sources through to the event-based running of JS: the initial `onload` event marks the transition point after which the document is asserted to be fully loaded; before this event fires, the page may be inconsistent as critical resources in the page may not yet have loaded, or scripts may still be writing into the document stream.

Explicitly highlighting these pipeline stages leads to designing extension points in a *principled* way: we can extend the inputs accepted or the outputs produced by each stage, as long as we produce outputs that are acceptable inputs to the following stages. This is in contrast to

[7] We ignore for now web-workers, which are an orthogonal concern.

[8] Defining when pages are actually separate is non-trivial, and is a refinement of the same-origin policy, which in turn has been the subject of considerable research [7, 2]

```
public interface IDOMTagFactory {
  IEnumerable<Element> TagTemplates { get; }
}
```

```
public class HelloWorldTag : Element {
  string TagName { get { return "HelloWorld"; } }
  ...
}
public class HelloWorldFactory : IDOMTagFactory {
  IEnumerable<Element> TagTemplates { get {
    yield return new HelloWorldTag();
  } }
}
```

Figure 2: Factory and simple extension defining new tags

the extension models of existing browsers, which support various extension points without relating them to other possibilities or to the browser's behavior as a whole. The extension points engendered by the pipelines above are (as numbered):

1. Before beginning HTML parsing, extensions may provide *new tag names and DOM-node implementations* for the parser to support.

2. Before running any scripts, extensions may *modify the JS global scope* by adding or removing bindings.

3. Before inserting subtrees into the document, extensions may *preprocess* them using arbitrary C# code.

4. Before firing the `onload` event, extensions may declaratively inject new content into the nearly-complete tree using *overlays*.

5. Once the document is complete and events are running, extensions may modify existing event handlers using *aspects*.

6. Before beginning CSS parsing, extensions may provide *new CSS properties and values* for the parser to support.

7. Before computing layout, extensions may provide *new layout box types and implementations* to affect layout and rendering.

Some of these extension points are simpler than others due to regularities in the input language, others are more complicated, and others are as yet unimplemented. Points (1) and (5) are novel to C3. C3 does not yet implement points (6) or (7), though they are planned future work; they are also novel. We explain points (1), (3) and (4) in Section 3.1, points (2) and (5) in Section 3.2, and finally points (6) and (7) in Section 3.3.

3.1 HTML parsing/document construction

Point (1): New tags and DOM nodes The HTML parser recognizes concrete syntax resembling ⟨**tagName** attrName="val"/⟩ and constructs new DOM nodes for each tag. In most browsers, the choices of which tag names to recognize, and what corresponding objects to construct, are tightly coupled into the parser. In C3, however, we abstract both of these decisions behind a factory, whose interface is shown in the top of Figure 2.[9] Besides simplifying our code's internal structure, this approach permits extensions to contribute factories too.

Our default implementation of this interface provides one "template" element for each of the standard HTML tag names; these templates inform the parser which tag names are recognized, and are then cloned as needed by the parser. Any unknown tag names fall back to returning an `HTMLUnknownElement` object, as defined by the HTML specification. However, if an extension contributes another factory that provides additional templates, the parser seamlessly can clone *those* instead of using the fallback: effectively, this extends the language recognized by the parser, as XML3D needed, for example. A trivial example that adds support for a ⟨**HelloWorld**/⟩ tag is shown in Figure 2. A more realistic example is used by C3 to support overlays (see Figure 4 and below).

The factory abstraction also gives us the flexibility to support additional experiments: rather than adding *new* tags, a researcher might wish to *modify existing tags*. Therefore, we permit factories to provide a new template for existing tag names—and we require that at most one extension does so per tag name. This permits extensions to easily subclass the C3 DOM implementation, e.g. to add instrumentation or auditing, or to modify existing functionality. Together, these extensions yield a parser that accepts a superset of the standard HTML tags and still produces a DOM tree as output.

Point (3): Preprocessing subtrees The HTML 5 parsing algorithm produces a document tree in a bottom-up manner: nodes are created and then attached to parent nodes, which eventually are attached to the root DOM node. Compiler-authors have long known that it is useful to support *semantic actions*, callbacks that examine or preprocess subtrees as they are constructed. Indeed, the HTML parsing algorithm itself specifies some behaviors that are essentially semantic actions, e.g., "when an ⟨**img**/⟩ is inserted into the document, download the referenced image file". Extensions might use this ability to collect statistics on the document, or to sanitize it during construction. These actions typically are local—they examine just the newly-inserted tree—and rarely mutate

[9] Firefox seems not to use a factory; Chrome uses one, but the choice of factory is fixed at compile-time. C3 can load factories dynamically.

```
public interface IParserMutatorExtension {
  IEnumerable<string> TagNamesOfInterest { get; }
  void OnFinishedParsing(Element element);
}
```

Figure 3: The interface for HTML parser semantic actions

Base constructions	
⟨**overlay**/⟩	Root node of extension document

⟨**insert**
selector="*selector*"
where="before|after"/⟩
	Insert new content adjacent to all nodes matched by CSS *selector*

⟨**replace** selector="*selector*"/⟩	Replace existing subtrees matching *selector* with new content
⟨**self** attrName="value"…/⟩	Used within ⟨**replace**/⟩, refers to node being replaced and permits modifying its attributes
⟨**contents**/⟩	Used within ⟨**replace**/⟩, refers to children of node being replaced
Syntactic sugar	
⟨**before** …/⟩	⟨**insert** where="before"…/⟩
⟨**after** …/⟩	⟨**insert** where="after"…/⟩
⟨**modify** selector="*sel*" where="before"⟩ ⟨**self** *new attributes*⟩ *new content* ⟨**/self**⟩ ⟨**/modify**⟩	⟨**replace** selector="*sel*"⟩ ⟨**self** *new attributes*⟩ *new content* ⟨**contents**/⟩ ⟨**/self**⟩ ⟨**/replace**⟩ and likewise for where="after"

Figure 4: The overlay language for document construction extensions. The bottom set of tags are syntactic sugar.

the surrounding document. (In HTML in particular, because inline scripts execute *during* the parsing phase, the document may change arbitrarily between two successive semantic-action callbacks, and so semantic actions will be challenging to write if they are not local.)

Extensions in C3 can define custom semantic actions using the interface shown in Figure 3. The interface supplies a list of tag names, and a callback to be used when tags of those names are constructed.

Point (4): Document construction Firefox pioneered the ability to both define application UI and define extensions to that UI using a single declarative markup language (XUL), an approach whose success is witnessed by the variety and popularity of Firefox's extensions. The fundamental construction is the *overlay*, which behaves like a "tree-shaped patch": the children of the ⟨**overlay**/⟩ select nodes in a target document and define content to be inserted into or modified within them, much as hunks within a patch select lines in a target text file. C3 adapts and generalizes this idea for HTML.

```
⟨overlay⟩
  ⟨modify selector="head" where="after"⟩
    ⟨self⟩
      ⟨style⟩
        li > #bullet { color:  blue; }
      ⟨/style⟩
    ⟨/self⟩
  ⟨/modify⟩
  ⟨before selector="li > *:first-child"⟩
    ⟨span class="bullet"⟩&bull;⟨/span⟩
  ⟨/before⟩
⟨/overlay⟩
```

Figure 5: Simulating list bullets (in language of Fig. 4)

Our implementation adds eight new tags to HTML, shown in Figure 4, to define overlays and the various actions they can perform. As they are a language extension to HTML, we inform the parser of these new tags using the `IDOMTagFactory` described above.[10] Overlays can ⟨**insert**/⟩ or ⟨**replace**/⟩ elements, as matched by CSS selectors. To support *modifying* content, we give overlays the ability to refer to the target node (⟨**self**/⟩) or its ⟨**contents**/⟩. Finally, we define syntactic sugar to make overlays easier to write.

Figure 5 shows a simple but real example used during development of our system, to simulate bulleted lists while generated content support was not yet implemented. It appends a ⟨**style**/⟩ element to the end of the ⟨**head**/⟩ subtree (and fails if no ⟨**head**/⟩ element exists), and inserts a ⟨**span**/⟩ element at the beginning of each ⟨**li**/⟩.

The subtlety of defining the semantics of overlays lies in their interactions with scripts: when should overlays be applied to the target document? Clearly overlays must be applied after the document structure is present, so a strawman approach would apply overlays "when parsing finishes". This exposes a potential inconsistency, as scripts that run *during* parsing would see a partial, not-yet-overlaid document, with nodes a and b adjacent, while scripts that run after parsing would see an overlaid document where a and b may no longer be adjacent. However, the HTML specification offers a way out: the DOM raises a particular event, `onload`, that indicates the document has finished loading and is ready to begin execution. Prior to that point, the document structure is in flux—and so we choose to apply overlays *as part of that flux*, immediately before the `onload` event is fired. This may break poorly-coded sites, but in practice has not been an issue with Firefox's extensions.

[10] We apply the overlays using just one general-purpose callback within our code. This callback could be factored as a standalone, ad-hoc extension point, making overlays themselves truly an extension to C3.

3.2 JS execution

Point (2): Runtime environment Extensions such as Maverick may wish to inject new properties into the JS global object. This object is an input to all scripts, and provides the initial set of functionality available to pages. As an input, it must be constructed before HTML parsing begins, as the constructed DOM nodes should be consistent with the properties available from the global object: e.g., `document.body` must be an instance of `window.HTMLBodyElement`. This point in the document's execution is stable—no scripts have executed, no nodes have been constructed—and so we permit extensions to manipulate the global object as they please. (This could lead to inconsistencies, e.g. if they modify `window.HTMLBodyElement` but do not replace the implementation of ⟨**body**/⟩ tags using the prior extension points. We ignore such buggy extensions for now.)

Point (5): Scripts themselves The extensions described so far modify discrete pieces of implementation, such as individual node types or the document structure, because there exist ways to name each of these resources *statically*: e.g., overlays can examine the HTML source of a page and write CSS selectors to name parts of the structure. The analogous extension to script code needs to modify the sources of individual functions. Many JS idioms have been developed to achieve this, but they all suffer from JS's *dynamic* nature: function names do *not* exist statically, and scripts can create new functions or alias existing ones at runtime; no static inspection of the scripts' sources can precisely identify these names. Moreover, the common idioms used by extensions today are brittle and prone to silent failure.

C3 includes our prior work [10], which addresses this disparity by modifying the JS compiler to support *aspect oriented programming* using a dynamic weaving mechanism to advise closures (rather than variables that point to them). Only a dynamic approach can detect runtime-evaluated functions, and this requires compiler support to advise all aliases to a function (rather than individual names). As a side benefit, aspects' integration with the compiler often improves the performance of the advice: in the work cited, we successfully evaluated our approach on the sources of twenty Firefox extensions, and showed that they could express nearly all observed idioms with shorter, clearer and often faster code.

3.3 CSS and layout

Discussion An extensible CSS engine permits incrementally adding new features to layout in a modular, clean way. The CSS 3 specifications themselves are a step in this direction, breaking the tightly-coupled CSS 2.1 specification into smaller pieces. A true test of our proposed extension points' expressiveness would be to implement new CSS 3 features, such as generated content or the flex-box model, as extensions. An even harder test would be to extricate older CSS 2 features, such as floats, and re-implement them as compositional extensions. The benefit to successfully implementing these extensions is clear: a stronger understanding of the semantics of CSS features.

We discovered the possibility of these CSS extension points quite recently, in exploring the consequences of making each stage of the layout pipeline extensible "in the same way" as the DOM/JS pipeline is. To our knowledge, implementing the extension points below has not been done before in any browser, and is planned future work.

Point (6): Parsing CSS values We can extend the CSS language in four ways: 1) by adding new property names and associated values, 2) by recognizing new values for existing properties, 3) by extending the set of selectors, or 4) by adding entirely new syntax outside of style declaration blocks. The latter two are beyond the scope of an extension, as they require more sweeping changes to both the parser and to layout, and are better suited to an alternate implementation of the CSS parser altogether (i.e., a different configuration of C3).

Supporting even just the first two extension points is nontrivial. Unlike HTML's uniform tag syntax, nearly every CSS attribute has its own idiosyncratic syntax:

```
font: italic bold 10pt/1.2em "Gentium", serif;
margin: 0 0 2em 3pt;
display: inline-block;
background-image: url(mypic.jpg);
...
```

However, a style declaration itself is very regular, being a semicolon-separated list of colon-separated name/value pairs. Moreover, the CSS parsing algorithm discards any un-parsable attributes (up to the semicolon), and then parse the rest of the style declaration normally.

Supporting the first extension point—new property names—requires making the parser table-driven and registering value-parsing routines for each known property name. Then, like HTML tag extensions, CSS property extensions can register new property names and callbacks to parse the values. (Those values must never contain semicolons, or else the underlying parsing algorithm would not be able to separate one attribute from another.)

Supporting the second extension point is subtler. Unlike the HTML parser's uniqueness constraint on tag names, here multiple extensions might contribute new values to an existing property; we must ensure that the syntaxes of such new values do not overlap, or else provide some ranking to choose among them.

Point (7): Composing layout The CSS layout algorithm describes how to transform the document tree (the *content model*) into a tree of boxes of varying types, appearances and positions. Some boxes represent lines of text, while others represent checkboxes, for example. This transformation is not obviously compositional: many CSS properties interact with each other in non-trivial ways to determine precisely which types of boxes to construct. Rather than hard-code the interactions, the layout transformation must become table-driven as well. Then both types of extension above become easy: extensions can create new box subtypes, and patch entries in the transformation table to indicate when to create them.

4 Evaluation

The C3 platform is rapidly evolving, and only a few extensions have yet been written. To evaluate our platform, we examine: the performance of our extension points, ensuring that the benefits are not outweighed by huge overheads; the expressiveness, both in the ease of "porting" existing extensions to our model and in comparison to other browsers' models; and the security implications of providing such pervasive customizations.

4.1 Performance

Any time spent running the extension manager or conflict analyses slows down the perceived performance of the browser. Fortunately, this process is very cheap: with one extension of each supported type, it costs roughly 100ms to run the extensions. This time includes: enumerating all extensions (27ms), loading all extensions (4ms), and detecting parser-tag conflicts (3ms), mutator conflicts (2ms), and overlay conflicts (72ms). All but the last of these tasks runs just once, at browser startup; overlay conflict detection must run per-page. Enumerating all extensions currently reads a directory, and so scales linearly with the number of extensions. Parser and mutator conflict detection scale linearly with the number of extensions as well; overlay conflict detection is more expensive as each overlay provides more interacting constraints than other types of extensions do. If necessary, these costs can be amortized further by caching the results of conflict detection between browser executions.

4.2 Expressiveness

Figure 6 lists several examples of extensions available for IE, Chrome, and Firefox, and the corresponding C3 extension points they would use if ported to C3. Many of these extensions simply overlay the browser's user interface and require no additional support from the browser. Some, such as Smooth Gestures or LastTab, add or revise UI functionality. As our UI is entirely script-driven, we support these via script extensions. Others, such as the various Native Client libraries, are sandboxed programs that are then exposed through JS objects; we support the JS objects and .Net provides the sandboxing.

Figure 6 also shows some research projects that are not implementable as extensions in any other browser except C3. As described below, these projects extend the HTML language, CSS layout, and JS environment to achieve their functionality. Implementing these on C3 requires *no hacking of C3* , leading to a much lower learning curve and fewer implementation pitfalls than modifying existing browsers. We examine some examples, and how they might look in C3, in more detail here.

4.2.1 XML3D: Extending HTML, CSS and layout

XML3D [14] is a recent project aiming to provide 3D scenes and real-time ray-traced graphics for web pages, in a declarative form analogous to ⟨**svg**/⟩ for two-dimensional content. This work uses XML namespaces to define new scene-description tags and requires *modifying each browser* to recognize them and construct specialized DOM nodes accordingly. To style the scenes, this work must modify the CSS engine to recognize new style attributes. Scripting the scenes and making them interactive requires constructing JS objects that expose the customized properties of the new DOM nodes. It also entails informing the browser of a new scripting language (AnySL) tailored to animating 3D scenes.

Instead of modifying the browser to recognize new tag names, we can use the new-tag extension point to define them in an extension, and provide a subclassed ⟨**script**/⟩ implementation recognizing AnySL. Similarly, we can provide new CSS values and new box subclasses for layout to use. The full XML3D extension would consist of these four extension hooks and the ray-tracer engine.

4.2.2 Maverick: Extensions to the global scope

Maverick [12] aims to connect devices such as webcams or USB keys to web content, by writing device drivers in JS and connecting them to the devices via Native Client (NaCl) [17]. NaCl exposes a socket-like interface to web JS over which all interactions with native modules are multiplexed. To expose its API to JS, Maverick injects an actual DOM ⟨**embed**/⟩ node into the document, stashing state within it, and using JS properties on that object to communicate with NaCl. This object can then transliterate the image frames from the webcam into Base64-encoded src URLs for other scripts' use in ⟨**img**/⟩ tags, and so reuse the browser's image decoding libraries.

There are two main annoyances with Maverick's implementation that could be avoided in C3. First, NaCl

Extensions	Available from	C3-equivalent extension points used
IE:		
Explorer bars		(4) overlay the main browser UI
Context menu items		(4) overlay the context menu in the browser UI
Accelerators		(4) overlay the context menu
WebSlices		(4) overlay browser UI
Chrome:		
Gmail checkers	`https://chrome.google.com/extensions/search?q=gmail`	(4) overlay browser UI, (5) script advice
Skype	`http://go.skype.com/dc/clicktocall`	(4) overlay browser UI, (2) new JS objects, (5) script advice
Smooth Gestures	`http://goo.gl/rN5Y`	(4) overlay browser UI, (5) script advice
Native Client libraries	`http://code.google.com/p/nativeclient/`	(2) new JS objects
Firefox:		
TreeStyleTab	`https://addons.mozilla.org/en-US/firefox/addon/5890/`	(4) overlay tabbar in browser UI, inject CSS
LastTab	`https://addons.mozilla.org/en-US/firefox/addon/112/`	(5) script advice
Perspectives	[16]	(5) script extensions, (4) overlay error UI
Firebug	`http://getfirebug.com/`	(4) overlays, (5) script extensions, (2) new JS objects
Research projects:		
XML3D	[14]	(1) new HTML tags, (6) new CSS values, (7) new layouts
Maverick	[12]	(2) new JS objects
Fine	[6]	(1) HTML ⟨**script**/⟩ tag replacement
RePriv	[5]	(2) new JS objects

Figure 6: Example extensions in IE, Firefox, and Chrome, as well as research projects best implemented in C3, and the C3 extension points that they might use

isolates native modules in a strong sandbox that prevents direct communication with resources like devices; Maverick could not be implemented in NaCl without *modifying the sandbox* to expose a new system call and writing untrusted glue code to connect it to JS; in C3, trusted JS objects can be added without recompiling C3 itself. Second, implementing Maverick's exposed API requires carefully managing low-level NPAPI routines that must mimic JS's name-based property dispatch; in C3, exposing properties can simply *reuse* the JS property dispatch, as in Section 2.3.

Ultimately, using a DOM node to expose a device is not the right abstraction: it is not a node in the document but rather a global JS object like `XMLHttpRequest`. And while using Base64-encoded URLs is a convenient implementation trick, it would be far more natural to call the image-decoding libraries directly, avoiding both overhead and potential transcoding errors.

4.2.3 RePriv: Extensions hosting extensions

RePriv [5] runs in the background of the browser and mines user browsing history to infer personal interests. It carefully guards the release of that information to websites, via APIs whose uses can be verified to avoid unwanted information leakage. At the same time, it offers its own extension points for site-specific "interest miners" to use to improve the quality of inferred information. These miners are all scheduled to run during an `onload` event handler registered by RePriv. Finally, extensions can be written to use the collected information to reorganize web pages at the client to match the user's interests.

While this functionality is largely implementable as a plug-in in other browsers, several factors make it much easier to implement in C3. First and foremost, RePriv's security guarantees rely on C3 being entirely managed code: we can remove the browser from RePriv's trusted computing base by isolating RePriv extensions in an AppDomain and leveraging .Net's freedom from common exploits such as buffer overflows. Obtaining such a strong security guarantee in other browsers is at best very challenging. Second, the document construction hook makes it trivial for RePriv to install the `onload` event handler. Third, AppDomains ensure the memory isolation of every miner from each other and from the DOM of the document, except as mediated by RePriv's own APIs; this makes proving the desired security properties much easier. Finally, RePriv uses Fine [6] for writing its interest miners; since C3, RePriv and Fine target .Net, RePriv can reuse .Net's assembly-loading mechanisms.

4.3 Other extension models

4.3.1 Extensions to application UI

Internet Explorer 4.0 introduced two extension points permitting customized toolbars (Explorer Bars) and context-menu entries. These extensions were written in native C++ code, had full access to the browser's internal DOM representations, and could implement essentially any functionality they chose. Unsurprisingly, early extensions often compromised the browser's security and stability. IE 8 later introduced two new extension points that permitted self-updating bookmarks of web-page snippets (Web Slices) and context-menu items to speed access to repetitive tasks (Accelerators), providing safer implementations of common uses for Explorer Bars and context menus.

The majority of IE's interface is not modifiable by extensions. By contrast, Firefox explored the possibility that entire application interfaces could be implemented in a markup language, and that a declarative extension mechanism could *overlay* those UIs with new constructions. Research projects such as Perspectives change the way Firefox's SSL connection errors are presented, while others such as Xmarks or Weave synchronize bookmarks and user settings between multiple browsers. The UI for these extensions is written in precisely the same declarative way as Firefox's own UI, making it as simple to extend Firefox's browser UI as it is to design any website.

But the single most compelling feature of these extensions is also their greatest weakness: they permit implementing features that were never anticipated by the browser designers. End users can then install multiple such extensions, thereby losing any assurance that the composite browser is stable, or even that the extensions are compatible with each other. Indeed, Chrome's carefully curtailed extension model is largely a reaction to the instabilities often seen with Firefox extensions. Chrome permits extensions only minimal change to the browser's UI, and prevents interactions between extensions. For comparison, Chrome directly implements bookmarks and settings synchronization, and now permits extension context-menu actions, but the Perspectives behavior remains unimplementable by design.

Our design for overlays is based strongly on Firefox's declarative approach, but provides stronger semantics for overlays so that we can detect and either prevent or correct conflicts between multiple extensions. We also generalized several details of Firefox's overlay mechanism for greater convenience, without sacrificing its analyzability.

4.3.2 Extensions to scripts

In tandem with the UI extensions, almost the entirety of Firefox's UI behaviors are driven by JS, and again extensions can manipulate those scripts to customize those behaviors. A similar ability lets extensions modify or inject scripts within web pages. Extensions such as LastTab change the tab-switching order from cyclic to most-recently-used, while others such as Ghostery block so-called "web tracking bugs" from executing. Firefox exposes a huge API, opening basically the entire platform to extension scripts. This flexibility also poses a problem: multiple extensions may attempt to modify the same scripts, often leading to broken or partially-modified scripts with unpredictable consequences.

Modern browser extension design, like Firefox's Jetpack or Chrome's extensions, are typically developed using HTML, JS, and CSS. While Firefox "jetpacks" are currently still fully-privileged, Chrome extensions run in a sandboxed process. Chrome extensions cannot access privileged information and cannot crash or hang the browser. While these new guarantees are necessary for the stability of a commercial system protecting valuable user information, they also restrict the power of extensions.

One attempt to curtail these scripts' interactions with each other within web pages is the Fine project [6]. Instead of directly using JS, the authors use a dependently-typed programming language to express the precise read- and write-sets of extension scripts, and a security policy constrains the information flow between them. Extensions that satisfy the security policy are provably non-conflicting. The Fine project can target C3 easily, either by compiling its scripts to .Net assemblies and loading them dynamically (by subclassing the ⟨**script**/⟩ tag), or by statically compiling its scripts to JS and dynamically injecting them into web content (via the JS global-object hook). Guha et al. successfully ported twenty Chrome extensions to Fine and compiled them to run on C3 with minimal developer effort.

As mentioned earlier, C3 includes our prior work on aspect-oriented programming for JS [10], permitting extensions clearer language mechanisms to express how their modifications apply to existing code. Beyond the performance gains and clarity improvements, by eliminating the need for brittle mechanisms and exposing the intent of the extension, compatibility analyses between extensions become feasible.

4.4 Security considerations

Of the five implemented extension points, two are written in .Net and have full access to our DOM internals. In particular, new DOM nodes or new JS runtime objects that subclass our implementation may use protected DOM fields inappropriately and violate the same-origin policy. We view this flexibility as both an asset and a liability: it permits researchers to experiment with alternatives to the SOP, or to prototype enhancements to HTML and the DOM. At the same time, we do not advocate these extensions for web-scale use. The remaining extension points are either limited to safe, narrow .Net interfaces or are written in HTML and JS and inherently subject to the SOP. Sanitizing potentially unsafe .Net extensions to preserve the SOP is itself an interesting research problem. Possible approaches include using .Net AppDomains to segregate extensions from the main DOM, or static analyses to exclude unsafe accesses to DOM internals.

5 Future work

We have focused so far on the abilities extensions have within our system. However, the more powerful extensions become, the more likely they are to conflict with one another. Certain extension points are easily amenable to conflict detection; for example, two parser tag extensions cannot both contribute the same new tag name. However, in previous work we have shown that defining conflicts precisely between overlay extensions, or between JS runtime extensions, is a more challenging task [9] .

Assuming a suitable notion of extension conflict exists for each extension type, it falls to the extension loading mechanism to ensure that, whenever possible, conflicting extensions are not loaded. In some ways this is very similar to the job of a compile-time linker, ensuring that all modules are compatible before producing the executable image. Such load-time prevention gives users a much better experience than in current browsers, where problems never surface until runtime. However not all conflicts are detectable statically, and so some runtime mechanism is still needed to detect conflict, blame the offending extension, and prevent the conflict from recurring.

6 Conclusion

We presented C3, a platform implementing of HTML, CSS and JS, and explored how its design was tuned for easy reconfiguration and runtime extension. We presented several motivating examples for each extension point, and confirmed that our design is at least as expressive as existing extension systems, supporting current extensions as well as new ones not previously possible.

References

[1] Barth, A., Felt, A. P., Saxena, P., and Boodman, A. Protecting browsers from extension vulnerabilities. In *NDSS* (2010).

[2] Barth, A., Weinberger, J., and Song, D. Cross-origin JavaScript capability leaks: Detection, exploitation, and defense. In *SSYM'09: Proceedings of the 18th conference on USENIX security symposium* (Berkeley, CA, USA, 2009), USENIX Association, pp. 187–198.

[3] Bebenita, M., Brandner, F., Fahndrich, M., Logozzo, F., Schulte, W., Tillmann, N., and Venter, H. SPUR: A trace-based JIT compiler for CIL. In *OOPSLA/SPLASH '10: Proceedings of the 25th ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages and Applications* (New York, NY, USA, 2010), ACM.

[4] Fähndrich, M., Barnett, M., and Logozzo, F. Embedded contract languages. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing* (New York, NY, USA, 2010), ACM, pp. 2103–2110.

[5] Fredrikson, M., and Livshits, B. RePriv: Re-envisioning in-browser privacy. Tech. rep., Microsoft Research, Aug. 2010.

[6] Guha, A., Fredrikson, M., Livshits, B., and Swamy, N. Verified security for browser extensions. MSR-TR to be available 11/01, September 2010.

[7] Jackson, C., and Barth, A. Beware of finer-grained origins. In *In Web 2.0 Security and Privacy (W2SP 2008)* (2008).

[8] Jones, C. G., Liu, R., Meyerovich, L., Asanovic, K., and Bodík, R. Parallelizing the Web Browser. In *HotPar '09: Proceedings of the Workshop on Hot Topics in Parallelism* (March 2009), USENIX.

[9] Lerner, B. S., and Grossman, D. Language support for extensible web browsers. In *APLWACA '10: Proceedings of the 2010 Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications* (New York, NY, USA, 2010), ACM, pp. 39–43.

[10] Lerner, B. S., Venter, H., and Grossman, D. Supporting dynamic, third-party code customizations in JavaScript using aspects. In *OOPSLA '10: Companion of the 25th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2010), ACM.

[11] Meyerovich, L. A., and Bodik, R. Fast and parallel webpage layout. In *Proceedings of the 19th International Conference on the World Wide Web* (2010), WWW '10, pp. 711–720.

[12] Richardson, D. W., and Gribble, S. D. Maverick: Providing web applications with safe and flexible access to local devices. In *Proceedings of the 2011 USENIX Conference on Web Application Development* (June 2011), WebApps'11.

[13] Ruderman, J. Same origin policy for javascript, Oct. 2010.

[14] Sons, K., Klein, F., Rubinstein, D., Byelozyorov, S., and Slusallek, P. XML3D: interactive 3d graphics for the web. In *Web3D '10: Proceedings of the 15th International Conference on Web 3D Technology* (New York, NY, USA, 2010), ACM, pp. 175–184.

[15] Wagner, G., Gal, A., Wimmer, C., Eich, B., and Franz, M. Compartmental memory management in a modern web browser. In *Proceedings of the International Symposium on Memory Management* (June 2011), ACM. To appear.

[16] Wendlandt, D., Andersen, D. G., and Perrig, A. Perspectives: Improving ssh-style host authentication with multi-path probing. In *Proceedings of the USENIX Annual Technical Conference (Usenix ATC)* (June 2008).

[17] Yee, B., Sehr, D., Dardyk, G., Chen, J., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on* (May 2009), pp. 79 –93.