Featherweight Firefox

Formalizing the Core of a Web Browser

Aaron Bohannon
University of Pennsylvania

Benjamin C. Pierce
University of Pennsylvania

Abstract

We offer a formal specification of the core functionality of a web browser in the form of a small-step operational semantics. The specification accurately models the asynchronous nature of web browsers and covers the basic aspects of windows, DOM trees, cookies, HTTP requests and responses, user input, and a minimal scripting language with first-class functions, dynamic evaluation, and AJAX requests. No security enforcement mechanisms are included—instead, the model is intended to serve as a basis for formalizing and experimenting with different security policies and mechanisms. We survey the most interesting design choices and discuss how our model relates to real web browsers.

1 Introduction

Web browsers are complex: they coordinate network communication via many different protocols; they parse dozens of languages and file formats with flexible error recovery mechanisms; they render documents graphically, both on screen and in print, using an intricate system of rules and constraints; they interpret JavaScript programs; they manage users' browsing history, bookmarks, passwords, and RSS feeds; they execute cryptographic algorithms and protocols—and these are just the obvious tasks! This complexity makes it very challenging to design effective security mechanisms for web browsers: there are too many features to consider at once, and it is easy for the some of the fundamental security concerns to be obscured by more superficial problems.

"Web browser security" is actually an ambiguous term that can refer to a diverse range of issues. First, the browser codebase needs to be free from bugs such as buffer overflows that could lead to a complete compromise of a running browser. Next, the browser must correctly implement the protocols related to cryptography and the public key infrastructure for securing HTTPS communication and verifying digital signatures. To many users, "browser security" may be about how the interface helps them avoid phishing attacks and accidental installation of malware. And finally, a particularly interesting aspect of browser security is the restrictions that must be placed on web page scripts for the purpose of ensuring information security when managing documents and scripts from different sources. This last security problem is the one we are interested in addressing; we will refer to it as *web script security* to distinguish it from these other aspects of web browser security.

In current practice, web script security revolves around the idea of a *same-origin policy*. An origin is based on the domain name of a web page, and restrictions are placed on the interactions that can take place between pages with different origins. However, it is actually somewhat difficult to characterize the "policy" that these restrictions are intended to enforce; indeed, it is not even very clear which particular restrictions are meant by the phrase. Browsers implement origin-based restrictions on accessing the state of other windows, on navigating other windows to new locations, on accessing windows by their string name, on making AJAX requests, and on accessing and mutating cookies, but the idea of an "origin" is defined and used differently in each of these cases. For instance, whereas the ability to access other windows' state depends on the value of the `document.domain` property, the ability to make an AJAX request does not, and whereas some of these restrictions check that that the effective origins are identical, the ability for one window to navigate another depends on a complex set of rules about the windows' relationship and does not actually require that their origins be the same. Nonetheless, we will continue to use "same-origin policy" to refer to all of these restrictions.

Beyond these inconsistencies, the same-origin policy has other deficiencies. Whereas direct cross-domain interaction via window data structures and AJAX messages is disallowed, indirect cross-domain interaction by insertion of new document nodes that trigger HTTP requests is unrestricted. Furthermore, a script's origin is based solely on the URL of the page where it is placed rather than the URL from which the script was retrieved, and the origin of the data handled by scripts is never taken into account. Another problem is that the mechanisms that allow interaction between related domains (involving setting the `document.domain` property) are inflexible and potentially dangerous [8]. Moreover, many corner cases (especially those involving function closures) are missing in written specifications and are implemented differently in different browsers (see the work on Subspace [4] for an example). Finally, the same-origin policy offers no help in addressing the pervasive security problem of script-injection attacks, which are known as XSS ("cross-site scripting") attacks.

In comparison, what would an ideal proposal for web script security look like? There should be a clearly articulated policy that applies consistently to the web browser's operation as a whole. It should be flexible enough to guarantee rigid security boundaries similar to the same-origin policy when desired and also to permit cross-domain communication for mashup-like behavior if necessary. It should offer some account of XSS and CSRF[1] attacks. It should be written at a level of detail that makes it feasible to implement. Finally, it should be specified in such a way that claims about it can be verified through rigorous proof or model checking.

This last goal makes the task especially challenging: a formal specification of a security mechanism necessitates a formal specification of web browser behavior at some level of detail, and browsers are large systems with many interdependent components. However, the complexity of web browsers is exactly the reason it is important to write a rigorous specification! There has been confusion about the implementation of the relatively simple same-origin policy; if one is interested in security mechanisms that are even more sophisticated, it will be essential to pin down their behavior in a precise and formal manner.

How does one develop a formal model of a web browser? In theory, one could take a particular browser implementation to *be* the model. But this is too much detail: browser implementations are far too large to reason about formally. If we build a model with the right level of abstraction, we can study something of a more tractable size, and more importantly, we can focus on the fundamental aspects of web script security, putting aside the more superficial concerns until after the deeper logical issues have been ironed out.

Using the right level of abstraction cuts down on the size of a model, but there are still too many potentially security-relevant browser features for us to consider them all at once. We must begin with a core set of features. Once polices and mechanisms for these features are well understood, we can hope to extend the security mechanisms to cover additional features. What we have put in our initial browser model has been guided by our interest in the following aspects of the browser setting:

- The system works with data and code from many different principals, and security boundaries may need to take into account the author of the data, the author of the code, and the principal that caused the code to be run.
- Scripts can be downloaded at any time and are always run in an environment that is shared with other scripts. Therefore, dynamic evaluation of code is fundamental to the web browser scripting model.
- There are multiple top-level environments in which scripts can be run, and these environments change each time the user visits a new page.
- Scripts contain first-class functions, which are, among other things, used for event handling.
- The system is event-driven and events can be interleaved in complex and possibly unexpected ways.
- Network messages can be generated in a variety of ways by scripts and by the user; any message sent could be relevant to information security.

The primary contribution of the work presented in this paper is a formal specification of the core data structures and operations of a web browser. We use a small-step, reactive semantics that faithfully models the event-driven nature of web browsers. In cases where the appropriate browser behavior was not obvious, we referred to the HTML5 specification [3] and ran tests on browser implementations. At the end of Section 5, we discuss some of the differences between the HTML5 specification, browser implementations, and our model. Concretely our model is presented in the form of an OCaml program[2] (about 2,500 LOC, including many comments). The program is written in a form that corresponds very closely to logical rules of inference, which are commonly used in giving formal semantics to programming languages, and we can view it as a document to be read by researchers interested in formal browser semantics. Using OCaml as a concrete notation yields many benefits. First, the type system of the language gives a sanity check on the definitions. Second, the specification is executable, which facilitates testing and experimentation. (To this end, we have also written parsers for the relevant subsets of HTML, JavaScript, and HTTP.) The presentation here is structured around the main type declarations.

[1] A CSRF ("cross-site request forgery") attack occurs when a script on a page from one site generates an HTTP request resulting in an unwanted server-side transaction on another site.

[2] It can be accessed here: `http://www.cis.upenn.edu/~bohannon/browser-model/`

2 Key Concepts

We follow a top-down approach in describing our model. In this section, we survey the browser features that we included and we mention those we omitted. The remainder of the paper describes the specification itself, beginning from the outermost layer—the parts of a browser that are directly observable via its user interface and network connections—then moving on to internals.

Browsers display information in *windows*, many of which may be open at one time. A window can be *navigated* to a URL, which is the address of a document. The default URL for a newly opened window is "`about : blank`", which refers to a blank page. URLs that begin with "`http` :" refer to remote resources. When a window is navigated to such a URL, the browser sends an HTTP network request. The browser runs asynchronously and can handle other events while it is waiting for a network response.

For each URL that the user visits, browsers keep a mapping of key-value pairs known as *cookies*. These can be set each time an HTTP response is received, and the ones corresponding to a particular URL will be sent with each HTTP request for that URL. Including cookies allows us to model CSRF attacks; moreover, the unauthorized acquisition of session cookies is one of the most critical information security threats posed by XSS.

Browsers are designed to display HTML documents. For our purposes, it suffices to consider a subset of HTML with just a few basic features. It should allow tags to be nested in one another in some fashion. It should have some kind of text elements and link elements. It should also have text input elements and button elements that can be equipped with handlers, which are triggered by user interaction. Finally, it should have tags for including scripts in documents, both as source code that is written in-line in the document and as references to remote script files.

A document is transformed into a mutable document *node tree* and placed in a window. We use the term *page* to refer to a document node tree plus its related metadata, such the URL from which it was retrieved.[3] A window will display a sequence of pages over time as it is navigated to new locations.

Obviously, to study web script security, one must also have a model of the scripting language. The JavaScript language, which is used in current browsers, has many peculiarities and many of its peculiarities do have security implications, given the particular manner in which browser structures are represented in the JavaScript interpreter. However, instead of addressing problems that are specific to JavaScript at this time, we are more interested in understanding the security problems that are fundamental to *any* browser scripting language that can manipulate the browser in the ways that JavaScript can. This is a prerequisite step for understanding how to improve the security of a specific language such as JavaScript. Two of JavaScript's features are of interest to us because they seem especially useful in a web scripting environment and pose special challenges in the context of security: first-class functions and dynamic evaluation of code.

Although we want to restrict ourselves to a very simple core scripting language, we do want to capture a complete interface for scripting the browser components in our model. Scripts should be able to refer to their own window and refer to other windows by their name (which is just a string). They should be able to open, close, and navigate windows, as well as read and update their name. The name of a window is important to have in our model; it is relevant to security, both as a means of acquiring access to windows and as a means to transfer information. Scripts should be able to access a page's node tree and a page's global scripting environment through a reference to the page's window. We do not model the standard DOM in all of its details, but the scripting language should have enough power to construct and manipulate document node trees in arbitrary ways. Scripts should also be able to read and write the browser's cookies. Finally, scripts should be able to initiate AJAX-style HTTP requests and handle the responses.

Although our model encompasses many of the core features of a web browser, and certainly enough to make the issue of web script security challenging and interesting, there are many more features that we must leave out of our model for now, with the hope of considering them in the future. We do not consider relative URLs and fragment identifiers, although these would be fairly trivial. We do not consider virtual URLs schemes such as "`javascript` :" and "`data` :". We do not consider object-oriented features in the scripting language, nor do we consider object-style access to windows via the keyword `this` (instead we include the expression `self`, a synonym for `window`). We do not consider timer events, page load events, or low-level input events related to keystrokes and mouse movements. We do not consider any sort of frames, which offer a slightly different relationship between pages than having only separate top-level windows. We do not consider HTML forms nor browser history. We do not consider accessing files on the local machine. We do not consider any HTTP return codes other than "200 OK"; in particular, we do not consider HTTP redirects. We make no distinction between `http` and `https` URLs. We do not consider the interaction of web pages and scripts with plug-ins such as Flash or Java. Finally, we do not consider the password manager mechanism, nor any other browser extensions that

[3] We reserve the term *document* to refer to a static tree-like data structure, such as an HTML document.

might interact with web pages. We view all of these features as interesting and relevant to security, but we leave them out for now in order to minimize the complexity of our model.

Since our goal is to provide a foundation for researching new security policies and mechanisms, we designed our execution model as a "blank slate," without any security restrictions built in to it by default.

3 Reactive Systems

To begin with, we need to consider what the high-level "shape" of a browser's specification should be. A browser does not simply take an input, run for a while, produce an output, and then halt. Rather, it is an event-driven system: it consumes an input when one is available, which triggers some computation that may produce one or more outputs; when (and if) it is finishes running in response to one input, it will wait for the next input event. From the perspective of the scripts it runs, a browser should appear as if it has just a single event loop [3]. This sort of event-driven behavior can be captured by a fairly simple sort of state machine:

3.1 Definition: A *reactive system* is a tuple

$$(\mathit{ConsumerState}, \mathit{ProducerState}, \mathit{Input}, \mathit{Output}, \rightarrow)$$

where $\rightarrow$ is a labeled transition system whose states are $\mathit{State} = \mathit{ConsumerState} \cup \mathit{ProducerState}$ and whose labels are $\mathit{Act} = \mathit{Input} \cup \mathit{Output}$, subject to the following constraints:

- for all $C \in \mathit{ConsumerState}$, if $C \xrightarrow{a} Q$, then $a \in \mathit{Input}$ and $Q \in \mathit{ProducerState}$,
- for all $P \in \mathit{ProducerState}$, if $P \xrightarrow{a} Q$, then $a \in \mathit{Output}$,
- for all $C \in \mathit{ConsumerState}$ and $i \in \mathit{Input}$, there exists a $P \in \mathit{ProducerState}$ such that $C \xrightarrow{i} P$, and
- for all $P \in \mathit{ProducerState}$, there exists an $o \in \mathit{Output}$ and $Q \in \mathit{State}$ such that $P \xrightarrow{o} Q$.

Reactive systems never get "stuck," although they may get into a loop within the set of producer states and never accept another input. When a reactive system is in a consumer state, it must have some manner to handle whatever input comes along next, although it could choose to drop it and not do anything interesting in response. Reactive systems have a natural interpretation as a function from (possibly infinite) streams of inputs to (possibly infinite) streams of outputs. Bohannon, et al. [1] discuss these systems in more detail.

Given this template for a reactive system, what remains is to instantiate the system parameters—$\mathit{ConsumerState}$, $\mathit{ProducerState}$, Input, and Output—with the particular data structures that are relevant for web browsers. This will be done over the next two sections. What the system *does* is described by the transition relation of the reactive system, which we only have space to informally summarize in this paper as we describe the data structures; its full definition is given in the accompanying OCaml code.

4 Browser Inputs and Outputs

In this section, we present the structure of all of the data that goes into and comes out of a browser in our model. We describe these data structures using abstract syntax; for the purposes of an information security analysis, even user input and GUI updates can be modeled syntactically, as we will discuss in this section. We begin by looking at the abstract syntax for URLs, which is shown in Figure 1. We consider two kinds of URLs: a URL for a blank page and a basic `http` URL (without any port number or fragment identifier). The $\mathit{request\_uri}$ contains the path and query string of the URL.

The abstract syntax that we use for documents (see Figure 2) corresponds to an extremely simplified version of HTML. For comparison, the literal, concrete HTML syntax that would correspond to the abstract syntax is given on the right. (In order to literally translate our document constructs into well-formed HTML fragments, some must be mapped to HTML expressions with more than one tag, as shown.) Each construct in the abstract syntax has an optional string $\mathit{elt\_id}$, which should be thought of as the value of the `id` attribute of an HTML tag, if present. Unlike HTML, no further tags are allowed in the text of paragraph, link, or button elements. We append the suffix $\mathit{\_list}$ to syntactic categories as a way to indicate a sequence of zero of more items, such as the use of $\mathit{doc\_list}$ in the `div` construct.

The syntax of scripts is given in Figure 3. As with the browser as a whole, the goal of designing the scripting language is to capture the fundamental issues that make web script security interesting. We settled upon a very simple JavaScript-like core language—a dynamically typed language with mutable variables, a few basic data types, first-class functions, and dynamic evaluation—along with a set of constructs for manipulating the data structures of our browser model that is fairly complete in comparison with the standard "browser object model" (BOM) used in JavaScript. We didn't attempt to capture all of the idiosyncrasies of the BOM interface (the method for accessing cookies in JavaScript, for instance, is pointlessly absurd); however, we did aim to make the correspondence with the standard BOM very straightforward. We also did not try to aggressively eliminate redundant constructs since some constructs will likely need to be added, removed, and altered during the investigation of any particular security enforcement mechanism.

$$\begin{array}{rcl} \mathit{url} & ::= & \texttt{blank\_url} \\ & | & \texttt{http\_url}(\mathit{domain}, \mathit{request\_uri}) \end{array}$$

Figure 1: URL syntax.

$$\mathit{elt\_id} \quad ::= \quad \cdot \mid \mathit{string}$$

doc	::=	`para`(*elt_id*, *string*)	e.g., `<p>`*string*`</p>`
	|	`link`(*elt_id*, *url*, *string*)	e.g., `<p><a href="`*url*`">`*string*`</a></p>`
	|	`text`(*elt_id*, *string*)	e.g., `<p><input type="text" value="`*string*`"></input></p>`
	|	`button`(*elt_id*, *string*)	e.g., `<p><button>`*string*`</button></p>`
	|	`inline_script`(*elt_id*, *expr*)	e.g., `<script>`*expr*`</script>`
	|	`remote_script`(*elt_id*, *url*)	e.g., `<script src="`*url*`"></script>`
	|	`div`(*elt_id*, *doc_list*)	e.g., `<div>`*doc_list*`</div>`

Figure 2: Document syntax.

As in JavaScript, there are a variety of basic types (we do not consider classes, objects, or other user-defined types). The types `Null`, `Bool`, `Int`, `String`, and their corresponding literal expressions are straightforward. There is a type for URLs in the language, with corresponding literal URL expressions. In JavaScript, URLs are handled purely as strings, being parsed as needed; however, by having a special type for URLs in this language, we can avoid putting a string parsing algorithm in the main part of our semantics (such parsing could be done by a library function with a semantics specified completely separately). A value of type `Type` is a concrete representation of another value's type.

For compactness and uniformity, our language has no distinction between expressions and statements. Expressions can be sequenced with a semicolon (;), and the combined expression yields the result of the second expression when it is finished executing. If the first expression in a sequence results in an error, the second expression is not run. If the second expression results in an error, the effects of the first expression are still registered in the browser state. There is no need for a `return` construct when there is no distinction between statements and expressions. Conditionals and loops are standard. A `while` loop always evaluates to `null` when it terminates, as do other constructs with no sensible result. The $\mathit{primitive\_functions}$ include any pure functions whose semantics is independent of the web browser environment, such as functions to check the type of a value, to perform arithmetic operations, or to convert data types to and from strings. The specification of these operations is completely orthogonal to the problem of a browser specification and is therefore not included in our work.

Like JavaScript, this language has first-class, anonymous functions with local variables. However, in this language, functions always have exactly one parameter, and their local variables must be declared at the beginning of the function. Unary function application is denoted by $\mathit{expr}(\mathit{expr})$. Values of type `Code` represent a syntax tree for an expression. Any expression can be treated as a syntax tree by enclosing it with the `code` construct. `Code` values can be evaluated with the `eval` construct. (The expression will be evaluated in the environment that lexically encloses the `eval` expression.) As with URLs, having a special type for syntax trees differs from JavaScript (which passes strings to `eval`) but allows us to define a rigorous semantics for dynamic evaluation while putting aside the complex but uninteresting process of turning strings into expressions. Variables are dereferenced in the nearest (lexically) enclosing scope in which they are defined. If not defined elsewhere, a variable is dereferenced in the global scope of the script, which is the environment associated with the page in which the script was loaded. It is a runtime error to dereference a variable that is not defined in some enclosing scope. Variable assignment updates the variable (or function parameter) in the nearest enclosing lexical scope in which it is defined; if it is not defined in any enclosing scope, then the assignment will create a binding in the script's global scope.

The scripting operations that are specifically relevant to the web browsing environment are shown in Figure 4. The construct $\texttt{get\_cookie}(u, k)$ evaluates to the cookie value associated with the string key k for the URL u or evaluates to `null` if no such cookie is defined. The construct $\texttt{set\_cookie}(u, k, v)$ sets the cookie with key

$$
\begin{array}{rcl}
\mathit{type} & ::= & \texttt{Null} \mid \texttt{Bool} \mid \texttt{Int} \mid \texttt{String} \mid \texttt{Url} \mid \texttt{Type} \\
 & \mid & \texttt{Function} \mid \texttt{Code} \mid \texttt{Window} \mid \texttt{Node} \\
\mathit{expr} & ::= & \texttt{null} \mid \mathit{bool} \mid \mathit{int} \mid \mathit{string} \mid \mathit{url} \mid \mathit{type} \\
 & \mid & \mathit{expr};\ \mathit{expr} \\
 & \mid & \texttt{if}(\mathit{expr})\ \{\mathit{expr}\}\ \texttt{else}\ \{\mathit{expr}\} \\
 & \mid & \texttt{while}(\mathit{expr})\ \{\mathit{expr}\} \\
 & \mid & \mathit{primitive\_functions} \\
 & \mid & \mathit{function}(x)\ \{\texttt{var}\ x_1;\ \ldots;\ \texttt{var}\ x_n;\ \mathit{expr}\} \\
 & \mid & \mathit{expr}(\mathit{expr}) \\
 & \mid & \texttt{code}(\mathit{expr}) \\
 & \mid & \texttt{eval}(\mathit{expr}) \\
 & \mid & x \\
 & \mid & x = \mathit{expr} \\
 & \mid & \mathit{browser\_operations}
\end{array}
$$

Figure 3: Script syntax.

$$
\begin{array}{rcl}
\mathit{browser\_operations} & ::= & \texttt{get\_cookie}(\mathit{expr}, \mathit{expr}) \\
 & \mid & \texttt{set\_cookie}(\mathit{expr}, \mathit{expr}, \mathit{expr}) \\
 & \mid & \texttt{xhr}(\mathit{expr}, \mathit{expr}, \mathit{expr}) \\
 & \mid & \mathit{window\_operations} \\
 & \mid & \mathit{node\_operations}
\end{array}
$$

Figure 4: Browser operation syntax.

k for the URL u to the string value v. The construct $\texttt{xhr}(u, m, h)$ initiates an AJAX request to the URL u, sending the string m as the message body. The network response will be expected to contain a script, which will be passed in the form of a `Code` value as an argument to the handler function h (which may then run it with `eval` or dissect it in some other way using some of the primitive operations that we leave unspecified).

The scripting operations relating to windows and pages are shown in Figure 5. In JavaScript, there is a distinction between the `Window` object and `Document` object. As implemented in JavaScript, this distinction does not add any real expressive power, so we do not attempt to emulate it in our language. A window can hold only one page at a time; so a reference to a window is implicitly a reference to a page, although which page that is may change over time (if, say, the user navigates the window elsewhere). The important thing for one to understand is which window-accessible data can vary when the window's page changes. These include the window's location URL, the root node of the document tree in the window, and the global environment of the window.

The keyword `self` refers to the window that holds (or held) the page in which the script was loaded; the construct $\texttt{opener}(w)$ refers to the window from which w was opened. The construct $\texttt{named\_win}(s)$ evaluates to the window whose name is the string s or evaluates to `null` if there is no such window. A new window can be opened to a particular URL u using $\texttt{open}(u)$. A window with a name n can be opened to a URL u using $\texttt{open\_named}(u, n)$; however, if a window with that name is already open, then that window will be navigated to the URL u, and no new window will be opened. Both constructs for opening windows evaluate to the new (or navigated) window. A window can be closed with $\texttt{close}(w)$, and $\texttt{closed}(w)$ yields a `Bool` indicating whether or not w is a valid reference to an open window.

The URL of the document currently in a window w can be read using $\texttt{get\_location}(w)$, and a window can be navigated to a new URL u using $\texttt{navigate}(w, u)$. (In JavaScript, this is done by assigning a value to the `location` property of a `Window` object.) A window name can be read or updated using `get_name` or `set_name`. The root node of the document node tree in a window w can be read using $\texttt{get\_root\_node}(w)$ and can be set to a new node node using $\texttt{set\_root\_node}(w, \mathit{node})$. Every page has an associated environment that serves as the global environment for any scripts loaded into the page. For a page in a window w, the variables in the global environment can be read or updated using the constructs $w.x$ and $w.x = \mathit{expr}$.

$$
\begin{array}{rcl}
\mathit{window\_operations} & ::= & \texttt{self} \mid \texttt{opener}(\mathit{expr}) \mid \texttt{named\_win}(\mathit{expr}) \\
& \mid & \texttt{open}(\mathit{expr}) \mid \texttt{open\_named}(\mathit{expr}, \mathit{expr}) \\
& \mid & \texttt{close}(\mathit{expr}) \mid \texttt{closed}(\mathit{expr}) \\
& \mid & \texttt{get\_location}(\mathit{expr}) \mid \texttt{navigate}(\mathit{expr}, \mathit{expr}) \\
& \mid & \texttt{get\_name}(\mathit{expr}) \mid \texttt{set\_name}(\mathit{expr}, \mathit{expr}) \\
& \mid & \texttt{get\_root\_node}(\mathit{expr}) \mid \texttt{set\_root\_node}(\mathit{expr}, \mathit{expr}) \\
& \mid & \mathit{expr}.x \mid \mathit{expr}.x = \mathit{expr}
\end{array}
$$

Figure 5: Window operation syntax.

$$
\begin{array}{rcl}
\mathit{node\_operations} & ::= & \texttt{remove}(\mathit{expr}) \\
& \mid & \texttt{insert}(\mathit{expr}, \mathit{expr}, \mathit{expr}) \\
& \mid & \texttt{remove\_handlers}(\mathit{expr}) \\
& \mid & \texttt{add\_handler}(\mathit{expr}, \mathit{expr}) \\
& \mid & \ldots
\end{array}
$$

Figure 6: Window operation syntax.

The most interesting among the constructs related to document nodes are shown in Figure 6. Nodes have a graph structure derived from the fact that some nodes (`div` nodes in particular) can have children; all manipulation of nodes must maintain the invariant that the node graph is a forest. The construct $\texttt{remove}(\mathit{node})$ updates the node store so that node is removed from being the child of any `div` node or is removed from being the root node of any page, if either is applicable; the relationship of node and its children are unaffected by this, which means that neither node nor any of its descendants will be visible in any page after the operation. The construct $\texttt{insert}(\mathit{parent}, \mathit{child}, n)$ first removes the node child (just as if $\texttt{remove}(\mathit{child})$ had been evaluated) and then inserts child as the nth child of parent. It is an error (and nothing is mutated) if parent is not a `div` node, if parent has fewer than n children, or if parent is a descendant of child in a node tree. The construct $\texttt{remove\_handlers}(\mathit{node})$ removes all handlers from a text input or button node, and the construct $\texttt{add\_handler}(\mathit{node}, h)$ adds the function h as a handler for a text input or button node. When the value of a text input box is changed or when a button is pushed, each of the node's handler functions will be applied to the argument node and run.

We represent user interactions using the syntactic messages shown in Figure 7. We assume that a user refers to a window using a natural number representing it's relative age among the open windows, the oldest window being 0. In this way, we need not model an actual two-dimensional graphical display. There are several basic actions a user can take. The operations `load_in_new_window` and `load_in_window` represent cases where the user directs the browser to some URL—perhaps by typing in a URL, by selecting a bookmark, or by clicking on a link in a page.[4] The constructs `link_to_new_window` and `link_to_named_window` are used to represent cases where the user follows links that open in different windows due to the `target` attribute of the link. They both must include the window where the link was found as their first piece of data, since that window will be the deemed the "opener" of the new window. The user can also close windows, of course. The constructs `input_text` and `click_button` both take a window and a natural number, representing the position of the input box or button in the page. When these input events are received, they will trigger the handlers of the appropriate element.

There are four basic outputs that are visible to the user in this model. First, a new window can be opened. There is no data associated with the `window_opened` event because new windows are always created with an empty page having the URL `about : blank`. When pages are loaded or updated, there may be visible changes to the document rendered on the screen. There is a data type $\mathit{rendered\_doc}$ (not defined here) that captures the structure of the document node tree that is visible from the user interface. In the `page_loaded` and `page_updated` events, the entire document in the window is sent to the user, regardless of how much of it was actually changed. User output events play a rather non-obvious role in the model. For the purposes of an information security analysis, we assume that all browser inputs, including those from the network, are visible to the user; moreover, the

[4] If we included the HTTP `Referer` header in our model, we would need to distinguish between clicking a link and typing in a URL.

$$
\begin{array}{rcl}
\mathit{user\_window} & ::= & \texttt{window}(\mathit{nat}) \\
\mathit{user\_input} & ::= & \texttt{load\_in\_new\_window}(\mathit{url}) \\
 & | & \texttt{load\_in\_window}(\mathit{user\_window}, \mathit{url}) \\
 & | & \texttt{link\_to\_new\_window}(\mathit{user\_window}, \mathit{url}) \\
 & | & \texttt{link\_to\_named\_window}(\mathit{user\_window}, \mathit{string}, \mathit{url}) \\
 & | & \texttt{close\_window}(\mathit{user\_window}) \\
 & | & \texttt{input\_text}(\mathit{user\_window}, \mathit{nat}, \mathit{string}) \\
 & | & \texttt{click\_button}(\mathit{user\_window}, \mathit{nat}) \\
\mathit{user\_output} & ::= & \texttt{window\_opened} \\
 & | & \texttt{window\_closed}(\mathit{user\_window}) \\
 & | & \texttt{page\_loaded}(\mathit{user\_window}, \mathit{url}, \mathit{rendered\_doc}) \\
 & | & \texttt{page\_updated}(\mathit{user\_window}, \mathit{rendered\_doc})
\end{array}
$$

Figure 7: User I/O syntax.

browser operates deterministically, modulo the ordering of the inputs. So in a theoretical sense, the user always knows the complete browser state regardless of what user interface outputs are generated in the model. We also assume that no other principal can see these outputs, so they do not have much significance in developing confidentiality policies. However, they do become significant if we wish to develop and reason about integrity policies.

The network-related input and output events are shown in Figure 8. The `request` construct is a simplified version of an HTTP request. Our model does not distinguish between GET and POST requests since the difference has little impact on web script security. The *cookies* are key-value mappings, and an extra string can be sent as the body of the request, as would be done in a POST request. An output on the network consists of a *domain* and a *request*. We abstract away from the DNS name resolution process. We model a network connection with the domain name to which the connection was made and a natural number to distinguish between multiple connections to the same domain. We take 0 to be the oldest connection for which a response has not yet been received, 1 to be the next-oldest, and so on. As stated earlier, we do not model any HTTP responses other than "200 OK". We assume that the body of a response consists of either a well-formed document or a well-formed script.

5 Internal Browser Structures

We have seen the parts of the specification that describe how a browser interacts with its environment. Now we need to consider what internal bookkeeping is needed for a browser to operate. There are choices to make here. For example, one could choose to have document nodes maintain references just to their parents, just to their children, or to both. Our goal was to find a clear and succinct way to describe how a browser operates; usually (but not always) this seemed easiest to achieve by avoiding maintaining redundant information in the state. So, in the example of document nodes, we chose to have document nodes maintain references only to their children.

A browser's basic state is a tuple of six components as shown in Figure 9: it has stores for windows, pages, document nodes, activation records, and cookies, and a list of the open network connections. The syntactic elements with the suffix *_ref* are unique atomic identifiers that are generated freshly when new items are put into a store during the browser's execution; then they are used to refer to the associated data in the various stores. A *cookie_id* consists of a domain name, a path, and a string value representing the key.

The basic browser data structures are defined in Figure 10. The data for a window includes its name (an optional string) and a reference to its page. There is a `simple_window` structure for windows that were opened directly by the user, and an `opened_window` structure for windows that have a reference to the window from which they were opened. A window is considered to be open (and therefore visible to the user) iff a reference to the window appears in the window store of the browser. Pages contain their location, a reference to their root document node, a reference to the activation record containing their global environment, and a queue of scripts and of markers for not-yet-received scripts that will need to be executed in the future. If the browser is waiting idly for its next input, the script queue of every page in the browser will either be empty or will contain a marker for a not-yet-received script at its head; any scripts at the front of a queue that are in hand will be executed before the browser halts.

The structure of document nodes mirrors the structure of documents, except for a couple of details. First, there is an extra piece of data for `text` and `button` nodes, a

$$\begin{array}{rcl}
\textit{request} & ::= & \texttt{request}(\textit{request\_uri}, \textit{cookies}, \textit{string}) \\
\textit{network\_output} & ::= & \texttt{send}(\textit{domain}, \textit{request}) \\
\textit{response} & ::= & \texttt{doc\_response}(\textit{cookie\_updates}, \textit{doc}) \\
 & \mid & \texttt{script\_response}(\textit{cookie\_updates}, \textit{expr}) \\
\textit{network\_connection} & ::= & \texttt{connection}(\textit{domain}, \textit{nat}) \\
\textit{network\_input} & ::= & \texttt{receive}(\textit{network\_connection}, \textit{response})
\end{array}$$

Figure 8: Network I/O syntax.

$$\begin{array}{rcl}
\textit{window\_store} & ::= & [(\textit{window\_ref}_1, \textit{window}_1), \ldots, (\textit{window\_ref}_n, \textit{window}_n)] \\
\textit{page\_store} & ::= & [(\textit{page\_ref}_1, \textit{page}_1), \ldots, (\textit{page\_ref}_n, \textit{page}_n)] \\
\textit{node\_store} & ::= & [(\textit{node\_ref}_1, \textit{node}_1), \ldots, (\textit{node\_ref}_n, \textit{node}_n)] \\
\textit{act\_rcd\_store} & ::= & [(\textit{act\_rcd\_ref}_1, \textit{act\_rcd}_1), \ldots, (\textit{act\_rcd\_ref}_n, \textit{act\_rcd}_n)] \\
\textit{cookie\_store} & ::= & [(\textit{cookie\_id}_1, \textit{string}_1), \ldots, (\textit{cookie\_id}_n, \textit{string}_n)] \\
\textit{browser} & ::= & \texttt{browser}(\textit{window\_store}, \textit{page\_store}, \textit{node\_store}, \textit{act\_rcd\_store}, \textit{cookie\_store}, \\
 & & \quad \textit{open\_connection\_list})
\end{array}$$

Figure 9: Browser state.

value_list, which is their set of handlers. Second, both kinds of script nodes need a flag to know whether they have already been queued for execution on some page. A script node must only be queued for execution once, even if it is later moved. Finally, the `div` node keeps a list of *references* to child nodes instead of a list of the literal child data elements as is done in the *doc* data structure.

The type *act_rcd* is used for activation records, an important part of the script evaluation. They contain a *bindings* data structure, which is a mapping of variable names to fully evaluated expressions. Moreover, activation records for local scopes must keep a reference to their parent record for proper variable access and update.

The last component of a browser's data is its list of open network connections. A record must be kept of the domain to which each request was made, the resource that was requested, and enough data to properly handle the resource when it arrives. A `doc_connection` is expecting a response with a document, which will be used to build a new page in a window; so, a reference to that window must be recorded. A `script_connection` is expecting a response that pertains to a script node in some document node tree; in order to find the appropriate queue item for the script on the appropriate page, a reference to that script node must be kept. Given our other implementation choices, the *page_ref* data here is actually redundant information, but tracking it here simplifies our operational specification a bit. An `xhr_connection` is expecting a response that should be given to a specific handler function. The handler function, a *value*, is kept as part of the connection data structure, but the handler must run with some definition for the window `self`. Here we have an choice of recording this information using a *window_ref* or a *page_ref*. Either one would work, as long as we can ensure that an AJAX response is never run on a page other than the one for which it was intended. This is slightly easier to do by keeping track of the *page_ref*.

A few more data structures are needed to manage the small-step evaluation of script expressions in browsers (see Figure 11). The internal language of expressions (*iexpr*) is a slight extension of the external scripting language. It has a set of values that includes closures, a term that represents an error, and a term that represents an expression in a particular scope, which will arise when closures are applied during evaluation. Closures and scoped expressions refer to a static context that includes both an activation record and a window reference that will determine the evaluation of the keyword `self`.

A browser in a running state requires a queue of *tasks* in addition to the basic browser state. The task queue keeps track of the script expressions that the browser must evaluate before it can accept another input. A task comprises an internal expression paired with the window with respect to which it should be evaluated. A task could equivalently be associated with a page instead of a window. The association between windows and pages cannot change between the time a task is enqueued and when it is executed (this association can only change immediately after receiving a network response). Since this information is needed primarily for evaluating the `self` keyword in the expression, we choose the window instead of the page.

$$
\begin{array}{rcl}
\mathit{window} & ::= & \texttt{simple\_window}(\mathit{window\_name}, \mathit{page\_ref}) \\
 & | & \texttt{opened\_window}(\mathit{window\_name}, \mathit{window\_ref}, \mathit{page\_ref}) \\
\mathit{queued\_expr} & ::= & \texttt{known\_expr}(\mathit{expr}) \\
 & | & \texttt{unknown\_expr}(\mathit{node\_ref}) \\
\mathit{page} & ::= & \texttt{page}(\mathit{url}, \mathit{node\_ref}, \mathit{act\_rcd\_ref}, \mathit{queued\_expr\_list}) \\
\mathit{node} & ::= & \texttt{para}(\mathit{elt\_id}, \mathit{string}) \\
 & | & \texttt{link}(\mathit{elt\_id}, \mathit{url}, \mathit{string}) \\
 & | & \texttt{text}(\mathit{elt\_id}, \mathit{string}, \mathit{value\_list}) \\
 & | & \texttt{button}(\mathit{elt\_id}, \mathit{string}, \mathit{value\_list}) \\
 & | & \texttt{inline\_script}(\mathit{elt\_id}, \mathit{expr}, \mathit{bool}) \\
 & | & \texttt{remote\_script}(\mathit{elt\_id}, \mathit{url}, \mathit{bool}) \\
 & | & \texttt{div}(\mathit{elt\_id}, \mathit{node\_ref\_list}) \\
\mathit{act\_rcd} & ::= & \texttt{global}(\mathit{bindings}) \\
 & | & \texttt{local}(\mathit{bindings}, \mathit{act\_rcd\_ref}) \\
\mathit{open\_connection} & ::= & \texttt{doc\_connection}(\mathit{domain}, \mathit{req\_uri}, \mathit{window\_ref}) \\
 & | & \texttt{script\_connection}(\mathit{domain}, \mathit{req\_uri}, \mathit{page\_ref}, \mathit{node\_ref}) \\
 & | & \texttt{xhr\_connection}(\mathit{domain}, \mathit{req\_uri}, \mathit{page\_ref}, \mathit{value})
\end{array}
$$

Figure 10: Browser data structures.

$$
\begin{array}{rcl}
\mathit{iexpr\_context} & ::= & \texttt{context}(\mathit{window\_ref}, \mathit{act\_rcd\_ref}) \\
\mathit{value} & ::= & \texttt{closure}(\mathit{iexpr\_context}, x, x_1, \ldots, x_n, \mathit{iexpr}) \\
 & | & \texttt{win}(\mathit{win\_ref}) \\
 & | & \texttt{node}(\mathit{node\_ref}) \\
 & | & \cdots \\
\mathit{iexpr} & ::= & \mathit{value} \\
 & | & \texttt{error} \\
 & | & \texttt{scoped}(\mathit{iexpr\_context}, \mathit{iexpr}) \\
 & | & \cdots \\
\mathit{task} & ::= & \texttt{task}(\mathit{window\_ref}, \mathit{iexpr}) \\
\mathit{running\_browser} & ::= & \texttt{running}(\mathit{task\_list}, \mathit{browser})
\end{array}
$$

Figure 11: Internal expressions and running browser states.

Each kind of input will initialize the task queue in a different manner. When the user interacts with a button or text box, the task queue will be initialized with one task for each of the input control's handlers. When a network response to an AJAX request is received, the queue will be initialized with the corresponding `xhr_connection` handler. When a document is received, a page will be created along with its script queue, and the browser's task queue will be initialized with the items from the front of the page script queue that are ready for execution. Similarly, when a network response is matched with a `script_connection`, it will cause the appropriate page script queue to be updated, and then the ready items from the front of that queue will be transferred to the browser's task queue. If script nodes are inserted into a page as the browser executes tasks at the head of the task queue, additional tasks corresponding to those scripts will be enqueued at the back of the browser task queue, provided they are not blocked by an `unknown_expr` in their page's script queue.

Given the definitions thus far, we can now state explicitly how a browser forms a reactive system. The instantiations of *ConsumerState*, *ProducerState*, *Input*, and *Output* are given in Figure 12. There is nothing very interesting here, except for a couple of technicalities. According to the definition of a reactive system, there must always be exactly one output when stepping from a *ProducerState*; however, given our internal data structures, in some cases a single small-step of execution in our model may produce multiple outputs. Such a system can be trivially reduced to a formal reactive system by adding an output buffer to the producer state structure: if more than one output happens to be produced in a step of the original machine, the derived machine

$$
\begin{array}{rcl}
\mathit{ConsumerState} & ::= & \mathit{browser} \\
\mathit{ProducerState} & ::= & \mathit{running\_browser}' \\
\mathit{Input} & ::= & \mathit{user\_input} \mid \mathit{network\_input} \\
\mathit{Output} & ::= & \mathit{user\_output} \mid \mathit{network\_output} \mid \bullet
\end{array}
$$

Figure 12: Internal expressions and running browser states.

will remain in a producer state and release one output at a time over multiple steps. Thus $\mathit{running\_browser}'$ would be identical to $\mathit{running\_browser}$ but with an output buffer. On the other hand, since $\mathit{ProducerState}$ is technically required to produce an output on every step, we use the symbol $\bullet$ to represent a trivial, "silent" output, when there would otherwise be no output.[5]

6 Formalization Challenges

One particularly tricky issue is deciding exactly when scripts will get executed. The HTML5 specification recommends that there be three different queues of scripts that will get executed at different times, depending on whether script tags have a `defer` attribute, an `async` attribute, or neither. (This is motivated in part by the existence of JavaScript's `document.write()` method; we intentionally omitted such functionality from our specification because it introduces a huge complexity overhead while being a technique that should be avoided in modern web programming.) Moreover, HTML5 prescribes that remotely retrieved scripts should be executed after parsing finishes, whereas inline scripts should be executed "immediately." On the other hand, some browsers such as Firefox appear to nonetheless execute all scripts on a page in the order they appear, regardless of whether the scripts are inline or remote. We wanted to avoid the additional complexity associated with executing scripts midway through parsing; so we chose a behavior that was close to how Firefox behaves when all script tags have the `async` attribute set. This choice could be tweaked in the future, but the precise order of script execution seems unlikely to affect the design of security mechanisms.

Another tricky case is what to do when a user navigates away from a page but that page had some code that is still runnable, say, as a button handler in another page. When such a closure runs, it may attempt to access its global environment or to use the `self` construct. In the browsers we tested, if the window's new page is from the same origin as the old page, there are no difficulties in running such a closure. However, if the new page is from a different origin, some browsers raise errors that are presumably security-related. For instance, in recent versions of Firefox, simply evaluating the expression `self` in such a closure generates an error, even if no access of its properties or methods is attempted. This is an interesting corner case of the same-origin policy for which the correct restrictions are unclear and browsers vary widely in their behavior. However, since we are not modeling security restrictions, our specification raises no errors in such cases. A related situation occurs when a closure from a page remains executable after the page's window has been closed. It is not clear that security plays any role in this case; nonetheless, there are a variety of behaviors that can be observed in browser implementations. Some browsers will raise an error if the closure tries to evaluate the expression `self` and others will not; some browsers will purge the variables in the closure's global environment and others will not. We have chosen to allow `self` to be evaluated to a reference to the closed window and to leave the closure's environment intact. Our model does perform one sort of garbage collection, though: a page will be removed from the page store when the page's window is closed or the page is replaced by a navigation operation. Primarily, this is done to ensure that scripts received in network responses will not be executed on a page that is no longer visible. However, the page's associated nodes and activation records are left in their respective stores after the page is removed since these may be referenced elsewhere in the browser.

When a user navigates away from a page, there may also be outstanding AJAX requests that have not received a response. In this case, many browsers will call the state change handlers for the requests before leaving the page, as if the responses had come back with a dummy HTTP response code such as 0. Similarly, when a user closes a window, there may be outstanding AJAX requests for the page in the window. Some browsers call the handlers before closing the window, but others do not. Since we are not modeling HTTP error responses, we simply chose not to trigger the handlers in any of these situations.

7 Related Work

Our work was motivated by an investigation of the paper "Information-Flow-Based Access Control for Web Browsers" by Yoshihama, et al. [7]. Their work offers a reasonable outline of a browser formalization; our work

[5]Reactive systems that force each step to have an output are simpler to reason about and no less useful for studying information security [1].

is an attempt to fill in the concrete details of a more realistic browser model. Our model uses heap structures with references, which are important for understanding how information flows through a browser. Furthermore, our model adds first-class functions, activation records, and multiple global environments that are associated with page structures. In addition, we have broken the browser's behavior down into a small-step, event-driven semantics with a complete characterization of the possible inputs and outputs of the system.

There have been other efforts to formalize subsystems of a browser. Maffeis, et al. [5] have written a formal specification for the JavaScript language. Their work is in the same spirit as ours but nearly orthogonal in terms of its content, in that they consider all of the details of the JavaScript language itself without formalizing the language's integration with the browser. Gardner, et al. [2] have developed a formal specification for a literal subset of DOM Level 1 [6]. Their work is again in the same spirit as ours but strives for greater accuracy in a narrower domain. We did not attempt to implement a literal subset of the DOM specification (as they did) but instead specialized our node operations for the particular types of nodes in our document model. Moreover, they developed a full-blown compositional Hoare-logic semantics, whereas for our purposes a simpler operational specification is sufficient.

The Browser Security Handbook [8], published on Google's web site, provides thorough documentation of many different security-related behaviors that can be observed in different browsers. Our methodology has been first to try to understand how browsers would work *without* any security restrictions, thus offering a platform on which many different experiments with security enforcement mechanisms can be performed. Moreover, for our baseline, restriction-free semantics, we are not especially interested in documenting every observable browser behavior, but rather in having a single semantics that embodies a reasonable compromise between the different existing behaviors and specifications. The key point for us is to ensure that our formalization is close enough to real-world browsers so that experimenting with new security mechanisms on top of it will offer meaningful insight into how these designs would fare in reality.

The HTML5 [3] specification goes into a great deal of detail about browser behavior. In fact, it seems to be the only written account of many aspects of browser behavior. It covers many more features than we can put in our formal model at this time; however, as a specification written in English, it is necessarily somewhat imprecise, and it does not cover all of the corner cases involving integration with a scripting language, such as cases involving function closures. In contrast, our work is intended to be a platform for carrying out rigorous proofs.

8 Future Work

Our next goal is to use our formal model to experiment with concrete confidentiality and integrity policies. To start, this means designing a system of security levels and associating them with the input and output events of the model [1]. In concept, this is a straightforward process; however, there are different ways to do it, giving rise to different policies when combined with the requirement of reactive noninterference. In comparison with the constraints of the same-origin policy, noninterference-based polices will be more strict about cross-domain interactions over the network but more lax about cross-domain interactions that are confined within the browser. Policy design, which should include an account of declassification and server-guided policy customization, is an interesting topic, but even a basic policy has a wide range of enforcement techniques that we may wish to study—anything from globally removing operations from the scripting language to implementing a fine-grained tracking of information flow. Our preliminary investigations suggest that proving enforcement mechanisms sound with respect to policies will be challenging, given the size of our browser model, but that it is nonetheless feasible.

References

[1] Bohannon, A., Pierce, B. C., Sjöberg, V., Weirich, S., and Zdancewic, S. Reactive noninterference. In *Proceedings of the ACM Conference on Computer and Communications Security* (2009), ACM Press.

[2] Gardner, P. A., Smith, G. D., Wheelhouse, M. J., and Zarfaty, U. D. Local Hoare reasoning about DOM. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (2008), ACM Press.

[3] Hyatt, D., and Hickson, I. HTML 5. Tech. rep., W3C, 2010. `http://www.w3.org/TR/html5/`.

[4] Jackson, C., and Wang, H. J. Subspace: Secure cross-domain communication for web mashups. In *Proceedings of the International Conference on World Wide Web* (2007).

[5] Maffeis, S., Mitchell, J., and Taly, A. An operational semantics for JavaScript. In *Proceedings of the Asian Symposium on Programming Languages and Systems* (2008). See also: Dep. of Computing, Imperial College London, Technical Report DTR08-13, 2008.

[6] Nicol, G., Wood, L., Sutor, R., Apparao, V., Isaacs, S., Hors, A. L., Wilson, C., Champion, M., Robie, J., Byrne, S., and Jacobs, I. Document object model (DOM) level 1 specification (second edition). W3C working draft, W3C, Sept. 2000. `http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/`.

[7] Yoshihama, S., Tateishi, T., Tabuchi, N., and Matsumoto, T. Information-flow based access control for web browsers. *IEICE Transactions on Information and Systems E92.D*, 5 (2009), 836–850.

[8] Zalewski, M. Browser security handbook, Dec. 2009. `http://code.google.com/p/browsersec/wiki/Main`.