TreeHouse: JavaScript sandboxes to help Web developers help themselves

Lonni Ingram*† and Michael Walfish*
*The University of Texas at Austin †Waterfall Mobile

Abstract

Many Web applications (meaning sites that employ JavaScript) incorporate third-party code and, for reasons rooted in today's Web ecosystem, are vulnerable to bugs or malice in that code. Our goal is to give Web developers a mechanism that (a) contains included code, limiting (or eliminating) its influence as appropriate; and (b) is deployable today, or very shortly. While the goal of containment is far from new, the requirement of deployability leads us to a new design point, one that applies the OS ideas of sandboxing and virtualization to the JavaScript context. Our approach, called TreeHouse, sandboxes JavaScript code by repurposing a feature of current browsers (namely Web Workers). TreeHouse virtualizes the browser's API to the sandboxed code (allowing the code to run with few or no modifications) and gives the application author fine-grained control over that code. Our implementation and evaluation of TreeHouse show that its overhead is modest enough to handle performance-sensitive applications and that sandboxing existing code is not difficult.

1 Introduction

This paper is about TreeHouse, a system that allows Web applications to safely include—in the code delivered to the browser—third-party modules that are unaudited, untrusted, and unmodified. By *Web application*, we mean any Web site that includes JavaScript.

Many of today's Web applications are closely integrated with third-party code and in fact depend on the correctness of that code. For example, *frameworks* are now in wide use; these are JavaScript libraries that serve as application platforms by abstracting messy aspects of the browser's interface [13, 19, 30, 45, 63, 67]. As another example, sites selling advertising space today include scripts supplied by *ad networks*; these scripts are only supposed to display content from the ad networks, but they can hinder or harm the enclosing page (even if running in a frame, as we explain below and in Section 2.3). A third example is *widgets*: code supplied by an off-site service to invoke that very service (for example, [7, 56]). To perform its function, the widget needs read and write access to the enclosing page.

Adding to the helplessness of applications, the third-party code can change unilaterally. Applications often include frameworks by *hyperlinking elsewhere*: for low latency, some frameworks' code is hosted by Content Distribution Networks (CDNs).[1] Similarly, applications include ad scripts and widget scripts by hyperlinking to the ad network or widget implementer. All of these cases are analogous to a desktop application that dynamically links to a module running on someone else's computer!

Web applications, then, are taking the risk of adding large, opaque, third-party code to their trusted computing base [12]. Whether from malice or bugs, this code can compromise the privacy of data [8], the integrity of the enclosing page [26, 57], and the availability of the application [8, 52] (for instance, the script can make many HTTP requests, slowing the page's load time).

Given this situation, our high-level goal is a mechanism by which Web developers can contain and control included code, whether written by a third-party or the Web developers themselves. The question that we must answer is: what interface should this mechanism expose, and how should we implement it?

Of course, this question is not new (not even in the Web context); our point of departure is in adopting the following two requirements:

- *Make it work today (or failing today, shortly).* Previous projects are more tasteful than ours; indeed their principles have inspired this paper. But realizing those principles has required deployability compromises that we hope to avoid. Specifically, we wish to minimize (a) browser modification [10, 14, 29, 33, 37, 38, 43, 59, 62] or redesign [5, 9, 16, 24, 39, 47, 53, 60], as these require changes in the entire Web ecosystem; (b) development-time code changes [11, 20, 35, 41], as these often impose a performance cost and always require framework authors to rewrite their code; (c) runtime code changes [44, 46], as these either sacrifice the performance benefit of hosting framework code in a CDN, or else incur a large performance cost in the browser (Section 7 explains further); and (d) server configuration, such as domain names for each trust domain [28, 31, 32, 64], as this impairs deployability.
- *Allow controlled, configurable influence.* The mechanism should allow only the access and grant only the resources needed for the contained script to do its job: frameworks should be given access to all elements in the enclosing page, widgets should be given access only to the portions of the page they are concerned with, and ads should have no influence on the enclos-

[1] Unfortunately, application code cannot even compare the hyperlink's target to a known content hash, owing to the same-origin policy (§2.1).

ing page. We note that browsers' *iframe* mechanism fails this requirement since code in a frame can still leak data or consume scarce browser resources owned by the application (see Section 2.3 for more detail).

TreeHouse's high-level approach is to provide a *sandbox* in which a Web application can run guest JavaScript code. Sandboxing (or jailing) on hosts [21, 22, 36, 49, 51, 58] and in browsers [15, 61] is an elegant way to run legacy code inside a given context while giving that code little (or configurably limited) influence on that context. These works restrict the *machine code* and *system calls* that the sandboxed code executes. Our scenario, however, calls for configurable control over code that has been programmed to the *browser's JavaScript interface*. Thus, we borrow the top-level idea of sandboxing but need an interposition mechanism that understands JavaScript and the browser's API.

Unlike other work that provides such interposition [10, 14, 33, 37, 38, 43], TreeHouse requires no browser changes: it is implemented in JavaScript using browsers' current functionality. Specifically, it repurposes *Web Workers* (a feature in recent browsers in which a page can run a script in a separate thread) as containers to run *guest code*. It avoids modification in that code by *virtualizing* the principal interface to the browser (known as the Document Object Model or DOM; see Section 2.1).[2] TreeHouse exerts configurable control over guest code using an approach analogous to trap-and-emulate in the virtual machines context [4]: it interposes on privileged operations, permitting them as appropriate.

We have implemented and evaluated a prototype of TreeHouse. We ported a benchmark suite [1], a Tetris clone [50], and two frameworks [45, 67] to TreeHouse. Using existing code with TreeHouse requires modest effort. TreeHouse's relative overhead for DOM operations is high, but its absolute costs are tolerable (to human users). With a small amount of engineering work, our prototype could be made ready for actual production use.

TreeHouse has a number of limitations. First, its trusted computing base (TCB) includes the browser and thus is not small (though the TCB exposes a minimal interface, namely a virtualized interface to Web Workers; see Section 3). Second, despite our best efforts, the guest code sometimes needs minor restructuring; however, the required code changes are few and easy to make (see Section 6). Third, while Web Workers are available in recent browsers and expected to become ubiquitous, we are currently in a transition period (see Section 2.4).

There is a lot of related work, and we cover it in detail in Section 7. For now, we just note that no other work that we are aware of virtualizes the browser in a backward compatible way, requires no server or domain configuration, and protects against resource exhaustion attacks. The contributions of this work, then, are as follows:

- Applying the operating systems ideas of sandboxing, virtualizing, and resource management to JavaScript.
- The design of TreeHouse, which instantiates these OS ideas without browser modification.
- The implementation and evaluation of TreeHouse.

2 Background

This section explains the aspects of the Web browser ecosystem that are relevant to TreeHouse.

2.1 Some details of modern Web browsers

A Web page is a ***document*** composed of HTML markup, CSS styles, and JavaScript (JS) code. HTML describes the structure and content of the document, CSS describes its visual presentation, and JavaScript [18] adds dynamic behavior. Browsers provide an API through JavaScript, called the ***Document Object Model (DOM)***, which represents the page as a tree of nodes with methods and properties. Scripts within a Web page use the DOM to examine and change the page.

The browser also exposes an API through JavaScript that provides network access, multimedia capabilities, file access, asynchronous interrupts, and local storage. A notable class in this API is ***XMLHttpRequest (XHR)***, which allows a script to make an HTTP request. The browser restricts such requests, allowing them only to the document's ***origin***, a tuple of (scheme, domain, port).[3]

This restriction is part of the ***Same Origin Policy (SOP)***, whose purpose is to contain information leaks. Consider a user with the authority to get data from a restricted site. If such a user visits a site with malicious scripts that issue XHRs as the user to the restricted site, then, in the absence of the SOP, the browser would permit the XHRs; the scripts could thus wrongly extract data and send it to the malicious site. The SOP prevents such leaks by regarding each origin as a separate security principal and then preventing the browser from becoming a channel that leaks information among principals.

Because of this model, scripts in documents from the same origin may access the DOMs of each other's documents but not the DOMs of documents from any other origin. Perhaps confusingly, the SOP includes exceptions to this rule for some types of content. For example, a document is permitted to include and execute scripts from other origins. However, the origin that the cross-origin script is assigned by the browser is the origin of the including document, *not* the origin from which the script

[2] Others have virtualized this interface, in the browser [14, 20, 32, 40, 41] and on the server [3, 17], but with goals different from ours (§7).

[3] This tuple is drawn from the document's URL; for example, the origin corresponding to https://www.example.com:1234/foo/bar.html is (https, www.example.com, 1234).

was downloaded. For example, if a page from foo.com includes a script from bar.com, the browser allows the script to access content from foo.com but not bar.com.

2.2 JavaScript

The following properties of JavaScript help TreeHouse in its goal of isolating scripts. First, a script cannot create a reference to an arbitrary memory location: a script can access only objects that it creates itself and objects that the browser hands to it. Second, JavaScript as a language provides no facilities for I/O, meaning that, with the exception of covert channels, scripts can communicate outside their environment only by using the browser's API. Finally, as of version 5.1 of JavaScript, scripts can ***freeze*** properties of objects. Once a property is frozen, further attempts to assign or delete its value have no effect.

2.3 Frames

Browsers ship with a mechanism called iframes that are intended to create a logically separate entity within an enclosing page. However, iframes do not provide the isolation that one might want. First, iframes run in the same thread as their enclosing page. If code blocks in an iframe, the whole page blocks. Second, iframes can consume resource budgets that the browser imposes on the entire page. For example, browsers limit the number of in-flight XHRs (to any origin and in total), and misbehaving code can exhaust this limit (as has been observed [8]).

2.4 Web Workers

JavaScript is single-threaded and does not support preemption. Absent further mechanism, then, a script must break up compute-bound tasks, periodically returning control to the browser's event loop, or else the page becomes unresponsive. This limitation has motivated ***Web Workers***, a recent[4] browser feature that lets documents run scripts in a "separate parallel execution environment" [2]. In all cases that we are aware of (desktops, smart phones, etc.), these separate environments are preemptively scheduled processes or threads, as provided by the underlying operating system. For computations that admit parallelism, then, Web Workers allow application developers to write code in a threaded style.

Web pages can create an arbitrary number of workers; each gets its own JavaScript environment. The origin assigned by the browser to those workers is the origin of the document that created the worker, called the ***parent document***. A worker and its parent communicate using an asynchronous message-passing facility provided by the browser (called `postMessage`). Scripts are otherwise isolated: a script in a worker cannot import a reference to an object outside the worker or export a reference to an object inside the worker. Workers also do not have access to the DOM or most other browser resources. However, they can import scripts by URL, create child workers, and issue XHRs.

The goal of Web Workers was concurrency, but TreeHouse repurposes them, as described in the next section.

[4]Web Workers are part of the HTML5 specification [2] and are supported by the latest versions of all major browsers. Internet Explorer (IE) is a special case. As of this paper's publication, IE 10.0, which supports Web Workers, is in beta and expected to ship in 2012; when it does, IE's dominance is such that Web Workers will quickly be on a large majority of desktops.

3 Design of TreeHouse

3.1 Threat model and requirements

Threat model. Our threat model assumes that an honest user interacts with a Web application using an uncompromised and correct browser. The application is written by an honest ***author***. We will assume that the following content is correct and served from uncompromised Web servers that are part of the author's trust domain: a distinguished HTML page (called the ***host page***), a distinguished set of JavaScript, and any CSS and JavaScript directly included in the host page. The adversary can control the contents of any JavaScript, HTML or CSS that is downloaded from a server not under the author's control (for example, the adversary can supply the ad or framework code); this content is ***untrusted*** by the author, meaning that neither the author nor TreeHouse can depend on the correctness of this content. For prudence, an author aware of his imperfections (Dr. Jekyll) may wish to regard code that he himself wrote as being supplied by the adversary (Mr. Hyde), trusting only a minimal host page and the distinguished JavaScript.

Requirements. The design of TreeHouse is driven by the following requirements.

- *Isolate untrusted content.* Before anything else, TreeHouse needs a mechanism for *isolating* content. (We make this notion more precise below.) Such content would ideally have no impact on the execution of TreeHouse or on the rest of the application, and limited impact on their performance.
- *Interpose on untrusted content.* It is not sufficient for TreeHouse simply to isolate content. To perform useful work, content needs to communicate with the application, to affect the browser, and to consume browser resources. That is, the untrusted content may need to interact with the document's DOM, to gain access to cookies or files, to create Web Workers, and to consume outbound network requests and local storage. However, this impact needs to be controlled. Thus, TreeHouse must interpose on attempts by untrusted content to do useful work, to decide which attempts are permissible. This will allow TreeHouse to protect integrity (for example, by forbidding unauthorized

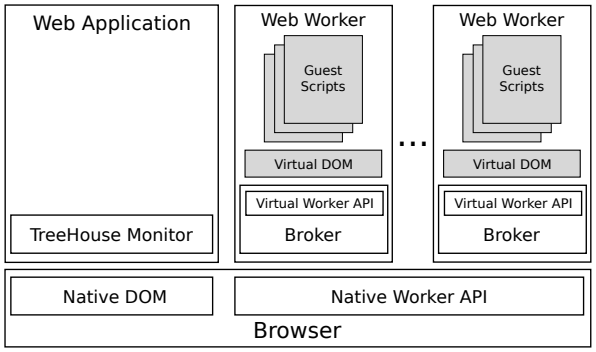

Figure 1—Architecture of TreeHouse. Shaded portions are untrusted by the application.

modifications to the DOM), availability (for example, by disallowing scripts from over-consuming required local storage), and privacy (for example, by disallowing a script from consuming an outbound network request to another origin, since it could use such a request to leak data).

- *Manage resources at fine grain.* TreeHouse must provide a way for application authors to express what access is permissible by guests, and what resources guests may consume. Consistent with the principle of least privilege [48], the granularity of permissions should be as fine as possible.

3.2 Overview of TreeHouse

Figure 1 depicts TreeHouse. For isolation, TreeHouse runs untrusted code in Web Workers (§2.4); once the code is running under TreeHouse in the Web Worker, we call it ***guest code*** or ***sandboxed code***. For interposition, TreeHouse installs a ***broker*** in each worker that virtualizes the browser's resources. For example, the broker exports to the worker a ***Virtual DOM (VDOM)*** that looks to guests like the browser's API. Interposition also requires a ***monitor*** that runs in the JavaScript environment of the window or tab in which the user loaded the application. The monitor applies guests' VDOM modifications to the real DOM, and delivers DOM events to guests—if permitted. What is permitted (regarding the DOM and access to other browser resources) is decided by the application author, and the definition of this policy is the only application customization; the monitor is the same across Web applications. Communication between the guest script and the monitor is handled by the broker, using message passing (§2.4). For example, the broker translates VDOM changes into messages to the monitor.

The rest of this section details the isolation mechanism (§3.3), interposition and virtualization (§3.4), and how application authors express policy (§3.5).

3.3 Isolation

We say that script *B* is ***isolated*** from script *A* if (1) *B* cannot prevent *A* from running; and (2) *B* cannot access *A*'s JavaScript environment. TreeHouse applies this notion to isolate untrusted scripts from the monitor and other application code. In the rest of this section, we describe how TreeHouse enforces (1) and (2). In short, TreeHouse uses Web Workers, which is perhaps surprising, since they were introduced for a different purpose.

For condition (1), each Web Worker runs in its own preemptively scheduled thread (§2.4), so the ability of a script inside a worker to affect the liveness of code outside it is restricted by the scheduling policy of the operating system. For example, if code in the worker enters an infinite loop, the performance of the system degrades but not to the point of preventing application code *outside* the worker from making progress.

For condition (2), we note that in a browser that correctly implements Web Workers (as assumed by our threat model), each worker has its own JavaScript environment. Moreover, JavaScript code cannot construct a reference to an object outside its environment (§2.2), and the only communication mechanism available to Web Workers, `postMessage` (§2.4), does not pass references.

Thus, to isolate a script, it is sufficient to run that script in a worker; this isolates the script from the monitor and from scripts in other workers.

3.4 Interposition and virtualization

We now motivate and describe TreeHouse's interposition mechanism. Interposition is needed for two reasons.

The first reason is that isolating code in the sense above does not prevent it from doing harm. Consider a Web application that allows users to upload and retrieve photos, and to send and receive private messages. Assume for illustration that these two functions are implemented by two logical services at the application's origin. If a script in the application is "supposed" to invoke only the photos API, then, by the principle of least privilege, that script should in fact be limited to the photos API. However, a malicious script—even if isolated—can still invoke the messages API and thus forge messages from the user. The script can also leak messages, as follows. Any origin can import scripts from any other (as discussed in Section 2.1 in the context of the SOP). If the malicious script has gained access to the user's private messages by invoking the messages API, the script can then exfiltrate a message by "importing" a script from http://evil.com/script.js?private-message, thereby leaking the private-message to evil.com.

To prevent attacks like the ones above and others, TreeHouse requires some logic between the isolated code and the outside world. This interposing logic must protect not only privacy but also integrity (by disallowing

unauthorized changes to the page seen by the user) and availability (by limiting the consumption of resources).

This brings us to the second reason that interposition is needed: some of the third-party actions need access to actual browser resources (the DOM, XHRs, etc.) to get work done. Thus, the interposing logic must not only intercept various actions but also decide which of them are permissible.

Questions of interface and mechanism. There are now three interrelated questions that TreeHouse must answer:

(1) Through what interface should guest code request resources from TreeHouse's interposing logic?

The challenge here is that scripts' resource requests need to be clear to TreeHouse (so it can decide whether to grant the request), yet the goal of deployability (§1) means that the script should not be altered to make requests through a new interface. TreeHouse's response to this challenge is reminiscent of trap-and-emulate [4] in the virtual machines context: TreeHouse *virtualizes* the browser's API and arranges to be invoked whenever a script requests access to a browser resource, whether that resource is one in the main application or in a Web Worker. But we now have a second question:

(2) What mechanism(s) should TreeHouse use to interpose and virtualize?

The challenge here is that JavaScript has a wide interface to the browser. This interface is narrower in Web Workers but still not so very narrow. Web Workers can access information about the application (such as its URL), import scripts, make network requests, and create child Web Workers. In some browsers, workers can use local storage as well as any files that the user has permitted the application to access. Meanwhile, TreeHouse needs to interpose on all of these actions. This brings us to the third question:

(3) How should application authors express policy so that TreeHouse can decide whether to grant or deny a given resource request?

The challenge here is ensuring a natural map between the language in which application authors express permissions and the implicit requests made by scripts. TreeHouse's approach here is to require application authors to express permissions in terms of the browser's API: the application author expresses which methods (and the arguments to those methods) guest code can use. Section 3.5 provides the details of policy expression. The rest of this section takes such a policy as a given and details TreeHouse's response to question (2).

Virtualizing the browser's API. Recall that TreeHouse's approach is reminiscent of trap-and-emulate. We first describe how TreeHouse arranges for traps and then how it performs the "emulate".

When TreeHouse creates a worker, it loads into the worker a script, called a ***broker***, which is part of TreeHouse's trusted computing base (see Figure 1). The broker must (a) interpose on calls to the browser's API that are available in workers (issuing XHRs, etc.) and (b) create a virtual DOM and interpose on interactions with it (the DOM itself is not available in workers; see Section 2.4). For (a), before the guest code loads, the broker modifies the worker's environment to wrap each function in the browser's API with a new implementation that interposes the broker when the function is called. Specifically, the broker uses JavaScript's `Object.defineProperty` API to associate the function name with a new function, and to freeze (§2.2) the association between the name and the new function, which prevents guest code from undoing this environment manipulation. For (b), the broker constructs a VDOM (§3.2), which contains subtrees of the real DOM (the application decides which subtrees). The VDOM implementation that TreeHouse uses (§5) raises events when the guest modifies the VDOM, and the broker registers a handler for such events.

For the "emulate" piece, there are two flows to consider: guest invocations of the browser API and event delivery from the main browser to the guest. When guest code invokes the browser's API or modifies the VDOM, the broker, being interposed, first applies the application's access control policy, to decide whether the guest action is permitted. If it is not permitted, the broker terminates the guest. If it is permitted, then, with the exception of DOM changes and asynchronous API methods for which a native implementation is not available in a Web Worker, the broker completes the call itself. We note that completing the call may involve further interposition—on the return value. For example, if the return value is an object with methods, then the broker replaces those methods with functions that interpose the broker.

In the case of DOM changes and asynchronous API methods, the broker delegates the request to the original browser API; to do so, the broker serializes the request and passes it to the monitor using `postMessage`. The monitor then makes the DOM modification or completes the API call. (We describe below how TreeHouse handles the mismatch between the synchronous interface to the VDOM and asynchronous `postMessages`.)

Event delivery is similar. If guest code wishes to register to be notified of a DOM event, then the broker receives the registration request and notifies the monitor. The monitor then registers its own *generic* handler in the application's DOM; when the event fires, the monitor notifies the broker, which re-raises the event in the VDOM on the handler registered by the guest.

As a final detail concerning virtualization, we note that the broker, when interposed, must sometimes do more

than check permissions. As an example, we briefly consider the API to create Web Workers. If the guest (itself in a Web Worker) attempts (and is permitted) to construct a Web Worker, the broker invokes the browser's API to construct a new Web Worker, and returns an object that wraps that new worker. To maintain interposition in the new worker, the broker, which is now a ***parent broker***, runs a ***child broker*** in the new worker. The child broker is identical in function to the parent except that, by default, it virtualizes only the interface that browsers expose in a Web Worker (no VDOM, etc.), as this is what a script that is *intended* to be run in a Web Worker would expect. The parent broker relays messages between the child broker and the monitor. TreeHouse virtualizes outbound network requests, file access, and local storage similarly.

A sync-vs-async mismatch, and some limitations. TreeHouse's approach to virtualizing the browser's API sometimes requires that the broker present a blocking interface; meanwhile, completing such a function or method call may require that the broker send an asynchronous `postMessage` to the monitor—and that the broker then return to the event loop so that it can receive the reply event (§2.4). The mismatch here is between a synchronous interface and an event-driven implementation, and there are two broad cases.

First, if the guest call can be "faked" by the broker, then the broker can present a synchronous interface and carry out the request asynchronously. For example, the broker presents a blocking interface to the VDOM and propagates changes to the application's DOM asynchronously. However, now TreeHouse must handle the equivalent of concurrent threads (the workers) sharing memory (the main DOM), where the threads have caches of that memory (the VDOMs in each thread). For simplicity, TreeHouse's response is to prevent sharing altogether: the monitor guarantees that a DOM node exists in at most one VDOM at once.[5] We do not believe that this limitation will be onerous for application authors.

The second case is when the guest call cannot be faked by the broker. As an example, consider `window.alert`, which creates a dialog box that blocks the calling script. No alert method is available in Web Workers, so for the broker to display an alert, it would have to send a request to the monitor and then return to the browser's event loop to await the reply—which conflicts with the guest's expectation of a blocking call. TreeHouse does not handle this case; if guest code calls such a method, it fails with a runtime error. Fortunately, there are few such methods, and they are rarely used by third-party code.

[5] An alternative approach would be to allow resources such as DOM nodes or cookies to appear in workers with either exclusive read-write access, or shared read-only access.

```
1  '!api': {
2   'XMLHttpRequest': {
3    '!invoke': true,
4    '!result': {
5     // permit only asynchronous XHRs
6     open: function (verb, url, async) {
7      return async === true;
8     },
9
10    '*': true // default rule
11   }
12  }
13 }
```

Figure 2—The portion of TreeHouse's base access control policy that governs XHRs. The policy forbids synchronous XHRs.

3.5 Resource control policy

The application author manages the guest's access to resources by (a) deciding which DOM elements to place in the guest's VDOM and (b) expressing policies that govern the guest's interaction with the browser's API (including the VDOM). This section details the second aspect; the next section gives examples of both aspects.

At a high level, the application author expresses access control policy in terms of the browser's API: what calls are permitted, and what arguments to those calls are permitted. In more detail, the author creates a per-guest JavaScript ***policy object*** and hands this object to TreeHouse (see Section 5 for the details of this hand-off). To simplify slightly, the policy object implements a key-value map from browser API elements to permissions: the keys name browser API elements,[6] and the values are ***rules***. A rule can be a Boolean value, a function, or a regular expression. If the rule is the Boolean value True, then the guest is permitted to invoke the given method or set the given property. If the rule is a function, the broker, at permission check time, executes the function (which should evaluate to a Boolean) to decide whether the action is permitted. If the rule is a regular expression, then it refers to a property; in this case, the guest is permitted to set the given property to a value v if v matches the regular expression.

TreeHouse has a ***base policy*** that authors are not supposed to override. This policy is there for their protection (and TreeHouse's). For example, as depicted in Figure 2, the base policy specifies that guests must open XHRs (§2.1) asynchronously; otherwise, a guest could prevent the broker from running. For a given action by a guest to take place, it must be permitted by *both* the per-guest policy and the base policy.

[6] The "keys" are structured hierarchically (reflecting the browser API's hierarchy), and can include wildcard components. This way, the author can use one rule to restrict the guest's use of an entire subtree of the hierarchy (say multiple API calls or elements).

```
1  var xhr = new XMLHttpRequest();
2
3  // initialize a request to get a list of the
4  // first ten photos
5  xhr.open('GET',
6   '/api/photos?start=0&count=10', true);
7
8  // register a callback to be notified when
9  // the XHR completes
10 xhr.onreadystatechange = function (e) {
11  if (xhr.readyState === 4) {
12   if(xhr.status === 200) {
13    console.log(xhr.responseText);
14   } else {
15    console.log("Error fetching photos");
16   }
17  }
18 };
19
20 xhr.send(null); // start the request
```

Figure 3—Example code to issue an XHR (§2.1) to a Web service that delivers photos.

```
1  '!api': {
2   'XMLHttpRequest': {
3    '!invoke': true,
4    '!result': {
5
6     // permit GET requests to resources on
7     // the application's origin whose URLs
8     // begin with '/api/photos'
9     open: function (verb, url, async) {
10     return verb === 'GET' &&
11       url.indexOf('/api/photos') === 0;
12    },
13
14    '*': true // default rule
15   }
16  }
17 }
```

Figure 4—Policy that gives limited permission: guest code can issue XHRs freely but only to particular services.

TreeHouse ships with a ***default*** or ***reference policy***, which whitelists unprivileged operations but denies everything else. For example, the reference policy forbids opening XHRs. This policy is 355 lines of code, including comments (and excluding unnecessary blacklisting for documentation). Overriding the reference policy need not be complex; we expect a typical policy to require 10–100 lines of code and no more than a few hours of work from the application author (see Section 6).

4 Examples

In this section we give several example uses of TreeHouse. Our first example illustrates how access control policies interact. We then describe containing advertisements and containing third-party widgets. Finally, we show how TreeHouse can protect a hypothetical plugin architecture (for example, by preventing exfiltration).

Limiting network access. Consider an author who—wishing to employ the principle of least privilege in the design of her application—breaks it into mutually distrusting components, each running in a TreeHouse sandbox. One such component is a script that displays a slideshow widget. The script obtains a list of photos to display from a Web service exposed on the application's origin and then renders the slideshow.

Figure 3 shows the guest's code for obtaining the list of photos. Because the default policy forbids constructing an XHR object, the code would fail at line 1. Thus, the author must override the default policy to give the guest code limited permission; Figure 4 depicts an example. Now when the broker intercepts the XHR constructor, it sees that `!api.XMLHttpRequest.!invoke` is True in both the guest policy and the base policy. The broker thus permits the construction and returns a wrapped XHR, as described in Section 3.4.

When the script calls the wrapped XHR's `open` method (lines 5 and 6 of Figure 3), that call succeeds, since the relevant policies permit the arguments to `open`. In more detail, when the script calls `xhr.open`, the broker checks the `!api.XMLHttpRequest.!result.open` property in the guest's policy object. The value of that property is a function (lines 9–12 of Figure 4), and TreeHouse calls it with the arguments provided by the guest script. The function returns True (because the URL is policy-appropriate), so TreeHouse then checks the base policy (lines 6–8 of Figure 2), which also returns True (because the guest has specified an `async` XHR).

The guest can also set `onreadystatechange` and call `send` (lines 10 and 20 of Figure 3) because the guest and base policies allow this through their default rule of `!api.XMLHttpRequest.!result.*`. Of course, the author could exert more fine-grained control by creating rules for properties and methods individually.

Containing advertisements. The provenance of advertisements (ads) is often murky: sites generally display ads by delegating a piece of their page to an ad network, which may populate the space or redirect to another ad network, and so on. Indeed, ads routinely misbehave [8, 57]. For these reasons, the best practice for a site selling space is to isolate an ad. Unfortunately, today's mechanism for such isolation, the iframe, does not eliminate attacks on availability (see Section 2.3). Under TreeHouse, in contrast, the application author can schedule and limit XHRs. For example, the author can associate `!api.XMLHttpRequest.!result.open` with

```
<script src="tetris.js"
 type="text/x-treehouse-javascript"
 data-treehouse-sandbox-name="worker1"
 data-treehouse-sandbox-children="#tetris"
></script>
<script src="tetris-policy.js"
 type="text/x-treehouse-javascript"
 data-treehouse-sandbox-name="worker1"
 data-treehouse-sandbox-policy
></script>
```

Figure 5—Example `script` tag. The depicted block specifies that `tetris.js` should run in a TreeHouse sandbox and that `tetris-policy.js` is the sandbox's policy.

a function that permits the method call only if the number of outstanding XHRs from the worker is below a given threshold. Or, given suitable hooks into the wrapping machinery, the application author can queue the `open` calls, sending them one at a time.

Containing widgets. Consider a set of widgets, each of which displays a one- to five-star rating next to an entry in a product list (e.g., [7]); such widgets are typically driven by a single script that runs in the application's page. For prudence, the application developer would like to limit the widgets' influence. (The developer cannot rely on iframes because, for reasons of layout, each widget would be in a separate iframe. With n products and thus n widgets, performance would suffer.) Under TreeHouse, the developer sandboxes the widget script and sets its VDOM to include only the locations in the document where the script should display ratings widgets. This pruned DOM can be constructed programmatically, using JavaScript functions that manipulate the DOM.

Avoiding exfiltration. Consider a webmail service, ExampleMail, whose authors want to allow third-party developers to create plugins. In the status quo, such plugins would be an unacceptable security risk, as the plugins would be able to read mail and then exfiltrate it. Under TreeHouse, ExampleMail's authors can sandbox a plugin and grant it limited access to the text of emails, while preventing it from exfiltrating email. Take, for instance, a plugin that displays the word count of the currently selected message in the user's inbox. This plugin receives a VDOM that includes the email that the user is viewing together with a *display element*, where the plugin author will display the word count. The application's policy prevents network access (by disallowing access to XHRs, WebSockets, and those attributes of DOM nodes that can have a URL as their value, such as `src`) and rejects DOM changes unless the change is to a node that descends from the display element. At that point, the plugin has access to the current message and can display the word count, but it cannot exfiltrate or alter the message.

method	**description**
start()	start the sandbox
terminate()	terminate the sandbox
addChild(node)	add a node to the VDOM
removeChild(node)	remove a child from the VDOM
addEventListener(type, function)	handle events from the sandbox
postMessage(message)	send a message to the sandbox
jsonrpcCall(method, args...)	make RPC call
jsonrpcNotify(method, args...)	make RPC call (no return value)
onPolicyViolation(function)	handle policy violations
setPolicy(policy)	set the access control policy

Figure 6—API for managing TreeHouse sandboxes.

component	lines of JavaScript
Monitor	350
Broker	349
Shared by monitor and broker	369
Access control policy	794
jsdom [17]	127,652

Figure 7—Lines of code in TreeHouse.

5 Integration and implementation

To integrate TreeHouse into a Web application, the author includes the TreeHouse monitor as traditional JavaScript inside the application page. There are two ways to sandbox a script. First, the author can include a `script` tag of type `text/x-treehouse-javascript` in the application's HTML, as illustrated in Figure 5. In this case, the browser creates a node for the script in the DOM but does not execute it (because it does not recognize the script type). When the browser loads the page, the monitor finds all such tags and creates sandboxes as appropriate. In this case, the author specifies the configuration options as attributes of the `script` tag; these options include which DOM subtrees the script may access, which worker to load the script into, and which policy applies to the script. The author's other choice is to invoke an API provided by TreeHouse, allowing the author to explicitly create a sandbox, set policy, etc. This API is depicted in Figure 6.

We have run TreeHouse successfully on Chrome, Safari, Firefox, and IE10. It is implemented in 1862 lines of JavaScript plus 127,652 for jsdom [17]. Jsdom is a *server*-side DOM implementation that we modified (with several hundred lines of code) to implement the VDOM. Figure 7 gives the breakdown (according to [42]). Besides jsdom, we use the underscore (v1.1.7) and RequireJS (v0.26.0) libraries for JavaScript utilities. We have not completed cookie virtualization or VDOM implementations of the new elements and API methods that the recent HTML5 standard introduces; this is future work.

	TreeHouse slowdown (×)			
benchmark	Chrome	Firefox	IE	Safari
dom-attr	32	39	9	—
dom-modify	15	72	21	120
dom-query	2600	8000	780	—
dom-traverse	7	14	1	12

Figure 8—TreeHouse overhead on Dromaeo DOM benchmarks reported as the geometric mean, over all benchmarks in a category, of TreeHouse's speed as a multiple of the baseline's. Empty entries result from browser incompatibilities or bugs. TreeHouse adds considerable relative overhead for DOM operations, but the absolute numbers are not high; see text.

	μ (σ) page load latency (ms)			
Experiment	Chrome	Firefox	IE	Safari
DOMTRIS, baseline	24 (8)	12 (1)	6 (3)	5 (1)
DOMTRIS, TreeHouse	361 (46)	181 (4)	405 (18)	166 (34)
118KB page, baseline	25 (3)	5 (1)	11 (5)	22 (5)
118KB page, TreeHouse	976 (38)	880 (18)	1229 (66)	779 (12)
Sandbox but no VDOM	350 (53)	136 (3)	323 (13)	132 (4)

Figure 9—Page load latency, with and without TreeHouse. TreeHouse's setup costs include a fixed cost from sandboxing and a cost for VDOM population that varies with VDOM size.

6 Evaluation

To evaluate TreeHouse, we answer two questions: (1) What is the latency overhead from TreeHouse? and (2) How easy is it to incorporate TreeHouse into an application? We answer both questions by experimenting with various benchmarks and Web applications.

Our experiments run on a MacBook Pro with a 2.66 GHz Intel Core 2 Duo processor and 4 GB of RAM, running Chrome 18.0.1025.168, Firefox 10.0.2, IE 10.0.8250.0, and Safari 5.1.5 (7534.55.3). We run Internet Explorer in a Windows 8 Consumer Preview (build 8250) guest on VirtualBox 4.1.12.

Benchmarks. The Dromaeo benchmark suite [1] is large (188 benchmarks) and diverse. Its benchmarks either do not access the DOM (*non-DOM benchmarks*) or else hammer it (*DOM benchmarks*). We expect that the non-DOM benchmarks would not experience significant slowdown under TreeHouse whereas the DOM benchmarks would run slower (because TreeHouse intercepts only DOM modification). To experiment, we run the suite with and without TreeHouse (which requires minor changes to Dromaeo, to report results via `postMessage`), performing 10 runs for each of the aforementioned browsers. Our expectations about the non-DOM benchmarks mostly hold: some benchmarks run slower (2–7×, depending on the benchmark and the browser) with TreeHouse, and some run faster, though we are still investigating to understand the slowdown and the speedup. For the DOM benchmarks, our expectations also hold; we now delve into those results.

To organize the DOM benchmarks, we divide them into four categories: *dom-attr*, which stresses setting attributes on DOM elements; *dom-modify*, which stresses inserting and removing DOM elements; *dom-query*, which stresses DOM searches; and *dom-traverse*, which stresses DOM tree traversals. Figure 8 reports the results, in terms of the geometric mean of each category's overhead. As expected, TreeHouse imposes significant overhead on DOM operations. The largest overhead (by far) is in DOM queries, and this overhead is staggering.

However, the operations that TreeHouse has blown up are not expensive in absolute terms or likely to be executed often. For example, depending on the browser, TreeHouse imposes overhead of 13,000–120,000 on `getElementsByTagName` (which returns all DOM nodes of a particular type, such as IMG) when searching for a non-existent type. Yet even after the blowup, each call to `getElementsByTagName` takes slightly less than 1 ms. Moreover, a best practice in Web application development is to avoid DOM queries (by caching). We expect applications that follow this practice not to be significantly slowed under TreeHouse. Of course, applications that do not follow this practice would need changes to run efficiently under TreeHouse.

Latency of page load. TreeHouse has two setup costs: *sandbox setup* (loading the guest into a worker, interposing the broker, etc.) and *VDOM population*. To assess both costs, we measure page load latency for DOMTRIS [50] (a JavaScript Tetris clone that uses the DOM to render the game and handle user input; the choice of application is borrowed from [40]) and a large Web page, with and without TreeHouse.

To measure the page load latency, we include two scripts, one in the page's header and one at the end of its body, measuring the time elapsed between each. When we measure under TreeHouse, the second script waits for a message from the guest, indicating that VDOM population is complete. This approach exploits the fact that the browser guarantees to execute the second script only after the entire DOM is parsed. We perform 10 runs in each browser, collecting timing data from JavaScript's `Date.now()`, which reports time in milliseconds.

Figure 9 reports the results with and without TreeHouse. For DOMTRIS, the average overhead of TreeHouse (TreeHouse row minus baseline row) is 161–399 ms, depending on the browser (here and below, the range reflects the minimum and maximum over the four browsers in our experiments). For the large Web page, the average overhead of TreeHouse is 757–1218 ms. We hypothesize that the larger overhead in this case derives from the size of the Web page's VDOM, which translates to higher VDOM population costs. To try to separate the sandboxing and VDOM costs, we run an experiment that

starts a TreeHouse sandbox with no VDOM; the average cost is 132–350 ms. TreeHouse's remaining costs come from application overhead and populating the VDOM.[7]

Usability for developers. TreeHouse requires two kinds of effort from developers: porting to TreeHouse and writing policies. We briefly assess each.

To port DOMTRIS to TreeHouse, we had to change only 28 lines of code in DOMTRIS. We are also in the process of porting several frameworks to TreeHouse; this effort both enhances TreeHouse's usability (by letting developers run existing framework-based applications in TreeHouse) and gives a sense of the work required to port a complex application to TreeHouse. So far, with 2 extra lines of code in the Zepto framework [67], all but three of the Dromaeo benchmarks that target Zepto's interface (consisting of 28 microbenchmarks) run successfully against Zepto-in-TreeHouse in Chrome and Safari. With changes to 18 lines of code in the Prototype framework [45], all but three of the Dromaeo benchmarks that target the Prototype interface (consisting of 29 microbenchmarks) run successfully against Prototype-in-TreeHouse in Chrome, IE, and Safari.

To evaluate the effort required to write real policies, we designed a sample policy for the ExampleMail plugin described in Section 4. The policy contains 41 lines of code and took approximately 30 minutes to write.

Summary. TreeHouse imposes significant overhead for "privileged operations", but, unless the application spends most of its time in such operations, total overhead should be far lower. TreeHouse also adds to initial page load latency, particularly when the VDOM is large. Porting applications to TreeHouse appears to require only modest effort, as does writing a non-trivial policy.

7 Related work

We survey related work by covering current browsers, isolation by frames, browser modification and redesign, language-based approaches, and related mechanisms.

Current browsers. The OP browser [24] creates a new process for each document. The Chrome browser, based on the Chromium project [5, 47], and Internet Explorer [65, 66] implement similar forms of isolation. These mechanisms, sometimes referred to as *process-per-tab*, allow the browser to contain site crashes and continue running. These mechanisms are orthogonal to TreeHouse: they isolate Web applications from each other but do not isolate scripts *within* a Web application.

[7] Our experiments do not let us determine the VDOM population costs precisely. We want to solve for V in $T = B + S + V$, where T is the TreeHouse results, B is the baseline results, and S is the sandboxing cost. Unfortunately, we observe T and B but not S. Instead, our experiment observes $S + E$, where E includes costs that B also includes. Thus one can derive a range for V, by varying E between 0 and B.

Frame isolation. Some schemes isolate JavaScript by using two browser features: the Same Origin Policy (§2.1) and iframes (§2.3). For example, SMash [31], Subspace [28], and OMOS [64] isolate scripts and components by running them in iframes served from different origins and then provide a mechanism for the iframes to communicate. AdJail [32] runs untrusted advertisement code in an iframe and then replicates changes to that iframe's DOM back to the main page.

All of these schemes contrast with TreeHouse as follows. First, unlike TreeHouse, they require the application to have a unique origin per isolated component. Second, as mentioned in Section 2.3, scripts in an iframe can interfere with the application's liveness, by going into an infinite loop or consuming resources; TreeHouse, in contrast, tolerates these cases (see Section 3.3). Third, these systems do not prevent the kind of information leaks detailed in Section 3.4 (their iframes can continue to "import" content from arbitrary locations).

Browser modification and redesign. We first summarize work that proposes browser re-architecture or modification and then explain why we avoided such changes.

The Atlantis browser [39] lets Web sites define their own layout, rendering, and scripting engines. Atlantis defines a small set of primitives and exposes them to sites. Applications can use Atlantis's primitives to achieve TreeHouse's goal: sandboxing scripts.

In contrast, other proposals for new browsers protect Web sites from each other but do not isolate scripts within a site. As examples, the Tahoma browser [9] isolates Web applications in virtual machines, the Illinois Browser Operating System (IBOS) [53] proposes an operating system and browser that map browser abstractions to hardware abstractions, and the Gazelle browser [60] treats origins as principals in a multi-principal operating system, isolating their content in restricted or sandboxed OS processes.

Some browser extensions have goals that overlap with those of TreeHouse. MashupOS [59] makes the browser a multi-principal operating system for Web applications. BEEP [29] lets Web sites restrict the scripts that run in each of their pages. ConScript [38] enforces application-specified security policies. OMash [10] restricts communication to public interfaces declared by each page. BFlow [62] adds information flow tracking to the browser, allowing untrusted JavaScript to operate on private data without compromising confidentiality.

The above projects partially inspire TreeHouse. However, they require browser changes. Meanwhile, if a browser change requires application changes (and all of the above proposals do), authors must either wait for all supported browsers to make the change or maintain two versions of their application. And an author who supports

an enhancement available in one or two browsers before it is adopted elsewhere takes the risk that other browser vendors choose not to add the feature at all. This risk is not small: browser vendors have historically been reluctant to implement new security features [27].

Language-based approaches. One way to isolate JavaScript is to constrain it to a subset of the language. Before this code is sent to the browser, it passes through a server-side verifier. Various projects apply this high-level approach, including Caja [41], FBJS [20], ADSafe [11], Browsershield [46], work by Maffeis et al. [34, 35], and work by Barth et al. [6]. JSReg [25] uses regular expressions to rewrite untrusted scripts.

The primary disadvantage of a restricted subset is that few libraries are written in them. Moreover, these subsets preclude many popular JavaScript idioms (e.g., the `this` keyword), making programming more difficult. While the designers of these approaches have gone to great lengths to ease the porting burden, the process of translating from arbitrary JavaScript to the restricted subset still requires manual work.

Language-based approaches have several other disadvantages. First, the application must serve the verified code from a server under its control. As a result, applications cannot gain from the performance advantages of content distribution networks (CDNs), as mentioned in the Introduction. Second, many language isolation techniques employ runtime logic that interposes on *all* object property accesses, which imposes non-trivial overhead. TreeHouse, by contrast, can isolate scripts from any origin and interposes only on privileged operations.

Related mechanisms. Several projects use Web Workers, virtualize the DOM, or create containers for isolated code. Some of these projects have inspired TreeHouse, but none shares all of our goals.

JSandbox [23] and Bawks [55] prototype a low-level IPC mechanism by which application code can load code into a Web Worker and allow callbacks. They do not provide virtualization (§3.4) or resource control (§3.5).

TreeHouse borrows DOM virtualization from previous work with different goals. Jsdom [17] (whose implementation TreeHouse uses) and dom.js [3] aim to provide a convenient toolkit for manipulating Web pages; they are geared to environments such as server-side JavaScript, where there is no "native" DOM. Mugshot [40] virtualizes part of the client's DOM but does so to create a replay system; it does not isolate scripts. AdSentry [14] is geared to isolation and, like TreeHouse, it interposes on DOM operations, applies access control policies, and delegates to the native DOM. Unlike TreeHouse, however, AdSentry requires browser modification and does not protect the application's liveness: if the ad blocks, so does the application.

In concurrent work, js.js [54] takes a different approach to sandboxing. The authors implement a JavaScript interpreter in JavaScript; the interpreter runs untrusted code and exposes no privileged methods or properties by default. While js.js and TreeHouse share some goals, the performance characteristics are different: DOM changes in js.js run at "native speed", but everything else runs two orders of magnitude slower than native. Under TreeHouse, DOM changes are expensive, but everything else runs roughly at native speed.

Finally, TreeHouse is inspired by classic approaches to isolation. Traditional sandboxing [21, 22, 36, 49, 51, 58], applied to browsers by Native Client [61] and Xax [15], contains legacy x86 code that expects to interact with the system call interface or machine resources. TreeHouse, however, contains legacy JavaScript code that expects to interact with the browser's resources.

8 Discussion and conclusion

We briefly consider how future help from browsers could address TreeHouse's limitations. First, the blocking-vs-event-driven mismatch and its consequences (§3.4) could be addressed if browsers exposed a way for JavaScript code to receive a message *synchronously*. Second, TreeHouse relies on the assumption that a worker cannot access the application's DOM; thus, it would be a boon to TreeHouse if browsers standardized the interface that is visible within workers. Of course, the standardization that would most address TreeHouse's performance and compatibility limitations would be incorporating TreeHouse—or functionality like it—into browsers.

Even if browser standardization does not come to pass, we believe that TreeHouse is promising: it is a practical, deployable, and usable way to give Web application authors fine-grained control over untrusted JavaScript code.

Acknowledgments

Insightful comments by John Hammond, Dave Herman, Jon Howell, Donna Ingram, James Mickens, Emmett Witchel, the anonymous reviewers, and our shepherd, Sam King, substantially improved this draft. This research was partially supported by AFOSR grant FA9550-10-1-0073 and by NSF grants 1055057 and 1040083.

TreeHouse is housed at `https://github.com/lawnsea/TreeHouse`. The site includes the code that implements TreeHouse, the pages used in our experiments, and demos.

References

[1] Dromaeo: JavaScript performance testing. `http://dromaeo.com/`.

[2] HTML5 living standard. `http://www.whatwg.org/specs/web-apps/current-work/multipage/`.

[3] A. Gal et al. dom.js. `https://github.com/andreasgal/dom.js`.

[4] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS*, 2006.

[5] A. Barth, C. Jackson, C. Reis, and the Google Chrome Team. The security architecture of the Chromium browser. http://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf, 2008.

[6] A. Barth, J. Weinberger, and D. Song. Cross-origin JavaScript capability leaks: Detection, exploitation, and defense. In *USENIX Security*, 2009.

[7] http://www.bazaarvoice.com/.

[8] J. Bixby. Fourth-party calls: What you don't know can hurt your site. . . and your visitors, July 2011. http://www.webperformancetoday.com/2011/07/14/fourth-party-calls-third-party-content/.

[9] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen. A safety-oriented platform for web applications. In *IEEE Symp. on Security & Privacy*, 2006.

[10] S. Crites, F. Hsu, and H. Chen. OMash: Enabling secure web mashups via object abstractions. In *ACM CCS*, 2008.

[11] D. Crockford. ADsafe: Making JavaScript safe for advertising. http://www.adsafe.org.

[12] Department of Defense. Trusted computer system evaluation criteria (orange book), 1985. DoD 5200.28-STD.

[13] Dojo Team. Dojo toolkit. http://dojotoolkit.org/.

[14] X. Dong, M. Tran, Z. Liang, and X. Jiang. AdSentry: comprehensive and flexible confinement of JavaScript-based advertisements. In *Annual Computer Security Applications Conference (ACSAC)*, 2011.

[15] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *OSDI*, 2008.

[16] J. R. Douceur, J. Howell, B. Parno, M. Walfish, and X. Xiong. The Web interface should be radically refactored. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2011.

[17] E. Insua et al. jsdom. https://github.com/tmpvar/jsdom.

[18] ECMA. *ECMA-262: ECMAScript Language Specification*, 5.1 edition, June 2011.

[19] Ext JS Team. Ext JS. http://www.sencha.com/products/extjs.

[20] Facebook Team. FBJS. http://developers.facebook.com/docs/fbjs/.

[21] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX Annual Technical Conference*, 2008.

[22] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *NDSS*, 2003.

[23] E. Grey. JSandbox. https://github.com/eligrey/jsandbox.

[24] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *IEEE Symp. on Security & Privacy*, 2008.

[25] G. Heyes. JSReg: JavaScript regular expression based sandbox. http://code.google.com/p/jsreg/.

[26] W. Huang. "HDD Plus" malware spread through major ad networks, using malvertising and drive-by download, Dec. 2010. http://blog.armorize.com/2010/12/hdd-plus-malware-spread-through.html.

[27] C. Jackson. Crossing the chasm: Pitching security research to mainstream browser vendors. http://www.usenix.org/events/sec11/stream/jackson/index.html.

[28] C. Jackson and H. J. Wang. Subspace: Secure cross-domain communication for web mashups. In *WWW*, 2007.

[29] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW*, 2007.

[30] jQuery Team. jQuery. http://jquery.com/.

[31] F. D. Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. SMash: Secure component model for cross-domain mashups on unmodified browsers. In *WWW*, 2008.

[32] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrishnan. AdJail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *USENIX Security*, 2010.

[33] T. Luo and W. Du. Contego: Capability-based access control for web browsers. In *International Conference on Trust and Trustworthy Computing*, 2011.

[34] S. Maffeis, J. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *IEEE Symp. on Security & Privacy*, 2010.

[35] S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *European Conference on Research in Computer Security*, 2009.

[36] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *USENIX Security*, 2006.

[37] L. A. Meyerovich, A. P. Felt, and M. S. Miller. Object views: Fine-grained sharing in browsers. In *WWW*, 2010.

[38] L. A. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE Symp. on Security & Privacy*, 2010.

[39] J. Mickens and M. Dhawan. Atlantis: Robust, extensible execution environments for web applications. In *SOSP*, 2011.

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

[41] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript, Jan. 2008. http://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf.

[42] CLOC: Count Lines of Code. http://cloc.sourceforge.net/.

[43] K. Patil, X. Dong, X. Li, Z. Liang, and X. Jiang. Towards fine-grained access control in JavaScript contexts. In *Intl. Conference on Distributed Computing Systems (ICDCS)*, 2011.

[44] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: Type-based verification of JavaScript sandboxing. In *USENIX Security*, 2011.

[45] Prototype Team. Prototype. http://www.prototypejs.org/.

[46] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *OSDI*, 2006.

[47] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *EuroSys*, 2009.

[48] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 63(9):1278–1308, Sept. 1975.

[49] M. Seaborn. Plash: tools for practical least privilege. http://plash.beasts.org/index.html.

[50] J. Seidelin. DOMTRIS: A DHTML Tetris clone. http://www.nihilogic.dk/labs/tetris/.

[51] C. Small and M. Seltzer. MiSFIT: Constructing safe extensible systems. *IEEE Concurrency*, 6(3):34–41, 1998.

[52] S. Souders. Performance of 3rd party content, Feb. 2010. http://www.stevesouders.com/blog/2010/02/17/performance-of-3rd-party-content/.

[53] S. Tang, H. Mai, and S. T. King. Trust and protection in the Illinois browser operating system. In *OSDI*, 2010.

[54] J. Terrace, S. R. Beard, and N. P. K. Katta. JavaScript in JavaScript (js.js): Sandboxing third-party scripts. In *USENIX WebApps*, 2012.

[55] P. Theriault. Bawks JavaScript sandbox. http://bawks.creativemisuse.com/.

[56] https://twitter.com/about/resources/widgets.

[57] A. Vance. Times web ads show security breach. *The New York Times*, page B5, Sept. 2009. http://www.nytimes.com/2009/09/15/technology/internet/15adco.html.

[58] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.

[59] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for Web browsers in MashupOS. In *SOSP*, 2007.

[60] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle Web browser. In *USENIX Security*, 2009.

[61] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *IEEE Symp. on Security & Privacy*, 2009.

[62] A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with BFlow. In *EuroSys*, 2009.

[63] YUI Team. YUI. http://yuilibrary.com/.

[64] S. Zarandioon, D. D. Yao, and V. Ganapathy. OMOS: A framework for secure communication in mashup applications. In *Annual Computer Security Applications Conference (ACSAC)*, 2008.

[65] A. Zeigler. IE8 and loosely-coupled IE (LCIE), 2008. http://blogs.msdn.com/b/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx.

[66] A. Zeigler. Tab isolation, 2010. http://blogs.msdn.com/b/ie/archive/2010/03/04/tab-isolation.aspx.

[67] Zepto.js Team. Zepto.js. http://zeptojs.com/.