Object Views: Fine-Grained Sharing in Browsers

Leo A. Meyerovich*
lmeyerov@eecs.berkeley.edu
UC Berkeley

A. Porter Felt†
apf@berkeley.edu
UC Berkeley

ABSTRACT

Browsers do not currently support the secure sharing of JavaScript objects between principals. We present this problem as the need for *object views*, which are consistent and controllable versions of objects. Multiple views can be made for the same object and customized for the recipients. We implement object views with a JavaScript library that wraps shared objects and interposes on all access attempts. Developers can control the fine-grained behavior of objects with an aspect system that accepts programmatic policies. The security challenge is to fully mediate access to objects shared through a view and prevent privilege escalation.

To facilitate simple document sharing, we build a policy system for declaratively defining policies for document object views. Notably, our document policy system makes it possible to hide elements without breaking document structure invariants. We discuss how object views can be deployed in two settings: same-origin sharing with rewriting-based JavaScript isolation systems like Google Caja, and inter-origin sharing between browser frames over a message-passing channel.

1. INTRODUCTION

Under current browser policies, sharing between principals is all or nothing. Since sharing everything can lead to cross-site scripting and related attacks, the browser security community has proposed many new ways to isolate principals from one another [16, 12, 28, 3, 25]. Given such isolation techniques, we explore the next problem: controlled sharing of resources between otherwise isolated principals [16, 5]. We present a mechanism for fine-grained mediation of shared objects.

Web principals have several resources worth sharing, such as their document, access to server-side data, and JavaScript APIs. Principals may want to share limited portions of these resources without giving up access to all of them. For example, consider an application that plots real estate prices on a map. Fine-grained sharing controls should let the application share only the map-relevant portions of the page with a third-party mapping service. The application and map service would then be able to exchange JavaScript objects and methods while maintaining separate internal invariants.

With our system, a principal can create a consistent, restricted wrapper for an object. We call the restricted version an *object view*. The restrictions might include, for example, making a property read-only or overriding a method so it always returns 0. Our object views support an *aspect system* [19] to implement these restrictions; we provide hooks on view actions so that programmatic policies can control the behavior of views. The most notable web resource we can wrap is the page document itself. Atop our aspect system, we build a declarative policy system for sharing document objects. We further show how to obscure document elements without breaking document traversal methods.

Our core mechanism for building views is a recursive wrapper. The security challenge is complete mediation: leaking references to unwrapped objects violates isolation, so views must avoid doing so. This is challenging because JavaScript is a flexible language; when passing object references across a trust boundary, even small gaps in the mediation strategy can enable privilege escalation [23, 20].

We apply our work to two web scenarios:

- **Same-origin sharing.** JavaScript rewriting systems [25, 17, 1] isolate gadgets from the rest of the page. To be useful (e.g., to isolate drawing on the screen), these systems must share heavily restricted DOM API access with gadgets. Current sharing protections are manual, requiring considerable maintenance and review.
- **Cross-origin sharing.** If browsers were to permit the sharing of objects, views could be used to mediate and restrict object access. We present a user-level implementation of object sharing atop the `postMessage` browser primitive by marshaling objects into strings and then controlling them with object views.

Our primary contributions are 1) the abstraction of an *object view* for fine-grained object sharing patterns, 2) the use of an aspect system to control shared object behavior, and 3) a discussion of how to build a JavaScript wrapper in an adversarial setting.

2. THREAT MODEL

We consider the security of object sharing between two web principals who are initially isolated from one another

*This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.
†Part of this work was done while the author was at Google Inc.

Copyright is held by the author/owner(s).
WWW2010, April 26-30, 2010, Raleigh, North Carolina.

by a trusted platform. We examine gadget aggregators and browsers as trusted platforms. Gadget aggregators isolate gadgets from one another and the surrounding page, and browsers isolate cross-origin frames from one another. We want to enable rich sharing across these boundaries without revealing additional privileges.

2.1 Web Security Model

Web pages have two primary components: static document content and JavaScript scripts[1]. Documents are represented by the Document Object Model (DOM), a hierarchical tree of page objects. Scripts are included with the original document or imported from a third party; imported scripts are given the same privileges as native scripts.

The current basic web security model is known as the Same Origin Policy (SOP). Under the SOP, a script in a document may access everything in that document and other documents from the same domain. All scripts in a document share one DOM and a set of global variables; pages in the same domain have separate sets of global objects, but additional references may be exchanged between them. A document's scripts do not have any access to documents from other domains.

2.2 Trusted Platforms

A *trusted platform* is responsible for separating principals from one another. Each *principal* is a script (or set of related scripts) that should have its own set of global objects and prototype chain. We examine two trusted platforms:

Server-Side Script Rewriters. Gadget aggregators (e.g., the Facebook Platform) want to embed third-party web applications directly into their own web sites. Here, the principals are the gadgets and the aggregator. Under the SOP, the browser does not provide any isolation guarantees between third-party applications and the aggregator site. Gadget aggregators therefore achieve isolation themselves with server-side tools that automatically verify and rewrite third-party scripts before they are added to the site [1, 25, 17]. Note that this is a one-way notion of security: the aggregator has full access to scripts, but scripts are restricted. Rewritten scripts cannot access global state unless a global object is specifically passed to them. Views are of interest to gadget aggregators because they need to provide gadgets with restricted versions of DOM nodes.

Browsers. Views could also be used to safely share objects between browser frames. Here, principals are frames. In this scenario, the browser provides isolation between the frames' principals as per the Same Origin Policy. Using views, principals could share simple objects or even versions of their documents. This could apply to both frames in separate windows and nested frames.

2.3 Attacks

In our object sharing scenario, we have two web principals separated by a trusted platform. One principal shares an object by creating an object view with a policy and sending the view to the other principal. Our attacker is the view recipient. The view sender intends to only share the object as restricted by the policy, but the attacker's goal is to steal additional privileges by manipulating the view in unexpected ways. For example, if Alice were to send Eve a completely unprotected DOM element, then Eve might be able to use that element to navigate up the DOM tree and access Alice's document cookie. We try to prevent privilege escalation by implementing a view as a wrapper around the shared object. An attacker will attempt to exploit weaknesses in our wrapper mechanism.

We identify four primary attack vectors that an attacker could use against a wrapper mechanism. To exemplify these vectors, consider this attempt to restrict an object. It is based on proposals by others [27, 32, 8]. The intended policy is to limit `postMessage` so that subsequent scripts can communicate with only the URLs specified by a whitelist:

```
<head><script>
(function () {
  var orig = frame1.postMessage;
  var wlist = {"msn.com": true};
  frame1.postMessage = function (m, url) {
      if (wlist[url])
        orig.apply(this, arguments); };
})();</script> ... </head>
```

The method is reassigned early into loading a page to prevent access to the original `postMessage` function. Wrapping the policy code in a function is an attempt to keep other scripts from accessing the `wlist` variable. However, this code is vulnerable to attack:

1. **Incomplete mediation.** The `postMessage` function is still accessible in other ways. The DOM and other libraries often provide multiple ways to perform an action, e.g., `frame2.postMessage.call(frame1, m, url)`.

2. **Unexpected parameter behavior.** We must defend against adversarial parameters. For example, consider this type forgery attack: Instead of passing a string for the parameter `url`, an attacker could pass an object with a malicious `toString` method that returns a different value each time it is invoked. Its first invocation occurs at `wlist[url]`, where it tricks the whitelist by returning a safe URL. The second invocation of `toString`, after `apply`, can return a different value not in the whitelist.

3. **Function prototype poisoning.** All JavaScript functions inherit basic properties and methods from the `Function` prototype. In this example, `orig.apply()` resolves to `Function.prototype.apply`. However, an attacker may maliciously reassign that prototype function to `eval`. After redefining `apply`, the attacker can add a forged entry to the whitelist:

   ```
   frame1.postMessage("wlist['m.com'] = 1", "msn.com")
   ```

4. **Object prototype poisoning.** All JavaScript objects inherit basic properties and methods from the Object prototype. The attacker could add a new field `Object.prototype.fake` and assign it to be true. When `"fake"` is passed as a URL parameter, `wlist["fake"]` will refer to the prototype property because the whitelist does not have its own property `"fake"`. Consequently, it will evaluate to true.

In Section 5, we describe how we systematically design our wrappers to defend against these attack vectors.

[1] A page may contain many types of scripts (e.g., Flash supports a JavaScript variant), but we only discuss JavaScript.

3. VIEWS FOR FINE-GRAINED SHARING

We present *object views*, a mechanism for securely sharing objects between principals. Views are used in place of the original object, and they support an aspect system. The aspect system lets developers install advice on views to control their behavior. We then build a declarative policy system on top of the advice system to make policy definition easier.

3.1 View Design

An *object view* proxies access to the original object and constrains access to it with a fine-grained policy. When Alice shares an object with Bob, she can create a view to share with Bob instead of the original object. We implement a view as a wrapper that enforces Alice's policy code on every attempt to access the object.

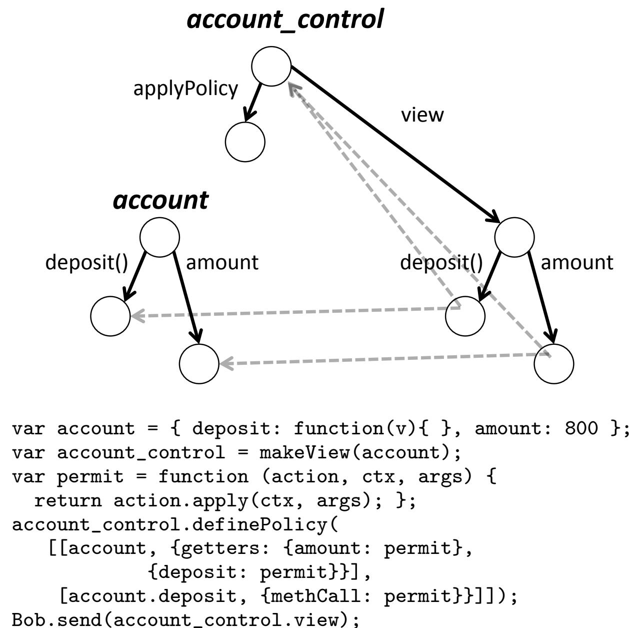
In Figure 1, Alice creates a view for her `account` object and gives it (`account_control.view`) to Bob using the communication primitive `send`. Calls through the view, if allowed, are proxied to the original object. Alice controls access to her view object by setting a policy, using `account_control.definePolicy`. As long as Alice does not share `account_control`, she is the only one who can control the view. In the example, she uses low-level abstractions to encode a behavioral whitelist: Bob may read the `amount` property and invoke the `deposit` method, but he cannot redefine either. Alice's access to the original `account` is not restricted, and Alice could make a different view for the same object by making a new control variable.

A view is a composition of a proxy and a policy. In our implementation, the proxy is a wrapper and the policy is a function. We create a new wrapper object for every object accessed through a view. A wrapper object is created by installing accessors (getter and setter methods) on the wrapper that correspond to all of the properties of the original object, including those inherited from the prototype chain. For each wrapped function object, we define a new proxy function object that calls a policy function instead of the original function. All operations on an object view go through a property accessor. Reading and invoking a method are different operations: assigning `v = obj.deposit` will trigger a getter, whereas invoking a method `obj.deposit()` will trigger a getter *and* run the proxy function. Running an unattached function object will only run a proxy function. A wrapper is applied recursively to return values. Note that a wrapped method can unwrap a wrapped parameter if the method and parameter are part of the same view.

Let us consider the property read `foo_v.bar.xyz`, where `foo_v` is a view of `foo`. First, the getter that we have assigned to the property `foo_v.bar` will be invoked. The getter will return a wrapped version of `bar`. Next, the getter that we have assigned to `bar_wrapper.xyz` will be invoked. If `xyz` is a primitive, then the getter will return that result. Otherwise, it will return a wrapped version of `xyz`.

For typical use, our wrappers are consistent with the original objects except where policies change view behavior. However, the level to which we can make our wrappers indistinguishable from original objects is limited by the semantics of JavaScript accessors; we cannot represent fields that are added after the view has been created.

We take care to preserve reference equality. If Bob repeatedly requests the same object from Alice, our system will repeatedly wrap it. If we were to create a new wrapper every time or reuse a wrapper from a different object,




```
var account = { deposit: function(v){ }, amount: 800 };
var account_control = makeView(account);
var permit = function (action, ctx, args) {
  return action.apply(ctx, args); };
account_control.definePolicy(
   [[account, {getters: {amount: permit},
            {deposit: permit}}],
    [account.deposit, {methCall: permit}}]]);
Bob.send(account_control.view);
```

Figure 1: Alice shares a view of her `account` object.

Bob would not have a consistent view of the object with respect to reference equality. Instead, we store a dictionary that associates wrappers and the objects they wrap so that we only generate a new wrapper for an object if it is not in this dictionary. Consequently, wrapping the same object twice yields the same wrapper each time, thereby preserving reference equality. We do not support reference equality in the presence of multi-principal delegation chains [24], referring to a scenario where an object is passed around between multiple parties.

3.2 Aspect System

Views support fine-grained policies. To accomplish this, we implement an aspect system [19]. The getters, setters, and proxy functions on wrappers provide interpositioning points for three *actions*: reading a field, setting a field, and calling a function. A policy author specifies an object and registers an advice function for each object action. *Advice* is a function that is applied "around" an object action. Advice might change arguments, choose not to perform the action, perform a different action, throw an exception, etc. An object's wrapper passes a property's advice function the original property or method, the `this` object, and any arguments. Unlike permit/deny policies, advice-based policies can significantly change behavior.

By default, we implement whitelisting: an action must be explicitly enabled. Whitelisting an action can be accomplished with the `permit` advice function shown in Figure 1. More sophisticated advice can perform actions like translation, encryption, and censorship. Advice can compare an argument to a whitelist or sanitize it to ensure it is purely alphabetic or numeric. For example, consider Alice sending Bob a Spanish-translated version of her object:

```
function translate (ctxt, fn, args) {
    return fn.apply(ctxt, args).toSpanish(); }
function say_hi () { return "hello"; };
```

```
var c = makeView(say_hi);
c.applyPolicy([say_hi, {funCall: translate}]);
bob.send(c.view);
```

Alice makes a view of her function object `say_hi` with advice that translates the result. Bob will see “hola” when he runs his view’s version of `say_hi`.

3.3 Document Sharing Policies

The DOM API provides scripts with access to a document’s structure and contents. It is large and complex, so programmatically expressing DOM policies as functions would likely be difficult and error-prone. To address this, we built a policy specification system that accepts declarative policies and translates them into advice.

Figure 2 presents an example policy that enforces read-only access to subtrees of a document. The policy author first specifies a collection of DOM elements and a set of restrictions to apply to these elements; in the example, the restrictions would apply to all elements with a class name “example”. Object interactions (read, write, “funCall” for function objects, and “methCall” for methods) can be associated with predefined advice (e.g., `permit`) or custom policy functions. Giving a method a “methCall” rule will by default also set `read: permit` for that property; this is not necessary for function objects. Every rule specifies the following:

1. *Selector.* An XPATH expression selecting a set of descendent nodes to apply the rule to.
2. *Enabled.* To allow any access to a node, the rule must specify that the node is enabled. We later will introduce the disabled state `obscure` as an alternative.
3. *Default and specific rules.* (Optional.) Default rules apply to all fields of the element. Specific rules such as `shake` in Figure 2 will apply to only the named field and have precedence over default policies. If multiple rules apply to the same element, all of them will be applied.
4. *Error.* (Optional.) Exception handler.

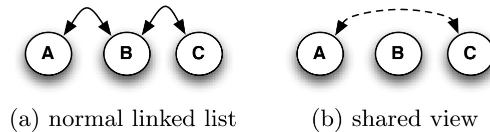
There are also policy-wide parameters, such as a default error handler.

Error-Free DOM Traversal. If done naively, restricting access to a document node could break expected invariants. Consider the task of disallowing all interaction with a single DOM element. If the view were to throw exceptions whenever that element is accessed, then the view recipient would experience unexpected exceptions while performing innocent tasks like iterating through the restricted element’s parent node’s children. We believe a better sharing policy would allow the view recipient to correctly navigate through the DOM tree even if an element is restricted.

```
var m = makePolicyView(makeView(document));
var policy =
 [{"selector": "(//*[@class='example'])
                 | (//*[@class='example']//*)",
   "enabled": true,
   "defaultFieldActions": {read: permit},
   "fields": {shake: {methCall: permit}}}];
m.applyPolicy(policy);
return m.view;
```

Figure 2: A policy with one rule that restricts subtrees to read-only if their root’s class name includes `example`. If a method `shake` exists, a more specific control list specifies that it may be invoked as well as read.

We address this need with a rule that *obscures* elements. An obscured element is not accessible, and we generate advice to prevent other DOM elements’ methods from returning references to the obscured node. Figure 3 demonstrates how a node in a linked list can be hidden by rerouting edges. Instead of specifying an `enabled:false` rule, the value `"obscured"` may be set. Our policy system then generates advice so that neighboring nodes’ traversal methods return the next node in the list instead of the obscured node. Our prototype does not yet handle the full DOM API; for instance, we do not yet change how `frame` elements are mirrored in the document-level frame array.



(a) normal linked list (b) shared view

Figure 3: A view hiding the existence of a linked list node.

Two peculiar cases arise when we discuss sharing the DOM in terms of views. First, actions on the view may have unclear mappings to raw objects. Consider Figure 3: it is unclear what should happen when a view recipient inserts a new node between `a` and `c`. On what side of node `b` should it be positioned in the original list? This scenario has an application-level solution: the view can be modified to expose a placeholder for `b`. A dual difficulty occurs when actions on the underlying object do not map clearly to the view. Suppose Alice shares a view that allows access to a tree but later obscures a part of it. If the view recipient has already stored a reference to the now obscured node, it is unclear what the parent pointer of the obscured node should be. For error-free hiding of nodes, an application should obscure a node upon its introduction.

4. VIEW-SHARING PLATFORMS

We examine how our view sharing primitive can be used with two different object sharing platforms: gadget aggregators with server-side script rewriting for same-origin sharing, and browser frames for cross-origin sharing. We do not limit the usefulness of views to these two scenarios; there are other settings (e.g., extensions and plug-ins) where partial DOM access and safe object sharing are desirable.

4.1 Server-Side Script Rewriting

Gadget aggregators use server-side script rewriters to isolate untrusted scripts from the rest of the page. Server-side script rewriters share two security goals:

1. Untrusted scripts should only be able to reference objects that they created or have explicitly been given access to. E.g., access to global variables is restricted.
2. DOM access must be protected. Scripts should only be able to access the DOM nodes that they created

or have been explicitly granted permissions for. They should also be restricted from API calls that would let them circumvent the aggregator's security policy (e.g., add uninstrumented JavaScript to the page).

In order to accomplish the second goal, script rewriting tools provide scripts with restricted versions of DOM nodes. For example, Google Caja [25] uses a set of handwritten DOM wrappers known collectively as Domita. The Caja developers have manually implemented a different wrapper for each node type. Their wrappers encode whether an object's fields are writable, readable, or invokable; and, if so, what arguments are acceptable. Unfortunately, the wrappers consist of thousands of lines of code (4111 lines of JavaScript in Caja as of this writing) and therefore require a significant amount of maintenance and review.

We advocate automatically generating views from declarative policies instead of handwriting wrappers. This approach more directly represents DOM policies and eliminates common attack vectors. While our current implementation focuses on DOM policies, our system could easily accommodate policies for other APIs (like the OpenSocial API) that an aggregator might want to share with a gadget.

An early version of our library was developed as a Caja library. Caja and similar rewriters provide more control over JavaScript features than vanilla JavaScript. For example, vanilla JavaScript does not provide a way to detect the addition of a new property to an object, which can cause a consistency bug between a view and the original object. However, Caja's accessors can be used to proxy an attempt to add a new field to a view.

4.2 Browser Frames

In this scenario, we use views to control cross-origin object sharing between frames. However, browsers do not currently provide a channel for reference passing across origin boundaries; this is the first work to propose safe object sharing between frames. Instead, browsers provide an inter-frame string passing mechanism named `postMessage`. Objects can be "shared" across this primitive by marshaling objects into strings and vice versa [2].

Although using `postMessage` means that no actual references are passed between windows, a marshaling library can still leak capabilities. Suppose that Alice implements a marshaling library that will carry out any operation on a shared object. Alice shares a form object `foo` with Bob, and then Bob asks the marshaling library for the value of `foo.parentNode.parentNode.parentNode.cookie`. This would reveal the value of the document cookie. If Alice had shared a view over the marshaling library, the view could enforce a policy that only whitelists `foo`'s safe properties. Object marshaling over `postMessage` has been proposed before [2]; our contribution is how to apply policies to objects shared over `postMessage` using views.

4.2.1 System Design

We present the use of views to safely share objects through a user-level `postMessage` marshaling library. Figure 4 shows views in use with a marshaling library. Sender Alice creates a view `tree_v` and restricts it with a policy as described in Section 3. The view creation library exists in sender Alice's frame because recipient Bob can manipulate any of the libraries in his frame, and Alice trust Bob. When Alice shares `tree_v` with recipient Bob, our sender library converts it

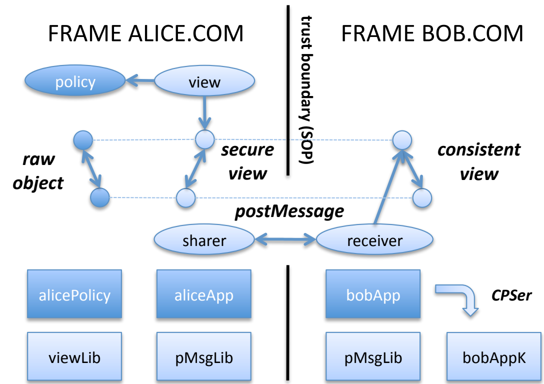


Figure 4: Cross-origin sharing diagram and source structure. Light elements are our JavaScript libraries, dark are developer-provided. We show the automatic CPS transformation of the recipient code.

to a string and sends it to Bob's frame using `postMessage`. Our recipient library receives the message via a callback and turns the string into an object to give to Bob. When Bob later requests `tree_v.prop`, our system forwards the property get request to Alice's frame, gets the result pursuant to the view's policy, and sends it back to Bob.

Figure 4 shows the basic one-way communication scenario, where the sender's state needs to be protected from an adversarial recipient. This is satisfactory in some scenarios, such as typical DOM interactions where DOM methods only accept primitives as arguments. However, Alice might share a function that accepts objects from Bob. If Bob wants protection from Alice, he should turn all of the non-primitive parameters that he passes to Alice into views as well. This could be automated by Bob's library so that all of his non-primitive parameters are converted to a view with a default policy, although our prototype does not implement this.

Marshaling Library. Our marshaling library performs all of the necessary tasks related to converting objects and setting callbacks so that developers do not need to do it themselves. The library initializes sessions, converts object description messages to objects, and keeps a cache to preserve reference equality for previously transmitted objects. Currently, the first level of an object is passed eagerly and the rest are passed lazily (i.e., an object's immediate child properties are transferred but not grandchildren). Our sharing and receiving libraries also encode the basic JavaScript operations of calling functions, getting and setting fields, and throwing exceptions. Potential future optimizations include sending further levels in anticipation of future requests, delay evaluation with *promises* [22], and manually delayed message bundles[2], but these present consistency and usability challenges.

Serialization systems are a potential point of attack. However, each message is a JSON object so we can rely on calls to standard browser support for JSON objects. The use of existing JSON APIs prevents unexpected code execution. We also want to ensure that our `postMessage` sender library does not leak additional information. This can be accomplished by not using any global state in the sender library: it only needs access to a view and a way to communicate with the intended primitive. The library can therefore only

provide access to the view.

It should be noted that functions and methods will always execute in the frame that defined the function or method. No code is ever passed between frames. When Bob asks to execute Alice's function `foo`, it will execute in Alice's frame and Bob will receive the result. Additionally, if Bob has redefined a method on an object of Alice's, all invocations of that method will execute in Bob's frame.

4.2.2 Asynchrony

`postMessage` is an asynchronous communication channel where messages are received through a callback. View access is therefore asynchronous in implementation, though developers often expect tasks like reading a field to be synchronous. Consider when Bob calls `alert(obj_v.prop)`. Retrieving the value of `obj_v.prop` is asynchronously made over `postMessage`, so there is no immediate value to pass to `alert`. One option would be to modify `postMessage` to be blocking; however, frames are single-threaded so this would freeze the frame. We want to stop just the *current* execution context while waiting on the call, allowing other events on the event loop to proceed. When the `postMessage` callback is invoked, the value of `obj_v.prop` becomes known and can now be passed to `alert`, etc.

Continuations are a well-established linguistic mechanism for saving the current execution context as a value that can be stored and later resumed. Currently, they are only directly supported by the Rhino JavaScript engine. In place of language support, we can translate source code into continuation passing style (CPS) using a client-side JavaScript library [7, 26]. Figure 4 shows Bob's code being automatically transformed. In CPS form, functions are called with an extra continuation argument that specifies what function should be called when the computed value is available (instead of directly returning the result). The transformed version of Bob's code would pass a continuation function to the getter on `obj_v.prop`; when the value is available, then the `postMessage` callback will invoke the continuation function with the result, thereby resuming execution.

JavaScript is typically written in a synchronous style, using callbacks only to react to user events and server events. However, some APIs are largely asynchronous because their methods do not have meaningful results. If an application only has a few points of inter-origin synchronicity, then a developer can manually CPS the program. Following the previous example, Bob would write `obj_v.prop(function(p) { alert(p); ... })` instead of `alert(obj_v.prop); ...` This approach does not scale well, however. Going forward, we advocate better language-level support for concurrency in browsers, even if not through continuations.

CPS does not remove all sources of concurrency errors induced by sharing objects over `postMessage`. If Alice takes a long time to respond to Bob, the user might trigger a GUI event in Bob's frame while Bob's code execution is suspended. Bob might have established an invariant before the call that he expects upon resumption, and the GUI event handler could violate this invariant. It might be possible to delay all events received while waiting for a response, but the GUI would appear unresponsive.

5. VIEW SECURITY

Wrappers enforce view security. The security goal of our wrapper mechanism is to ensure interposition on the shared object, which is equivalent to not leaking unwrapped references. As described in Section 2, we rely on trusted platforms to separate principals and provide each principal with its own private global objects and prototype chain. Here, we discuss how our wrappers defend against the attack vectors described in Section 2.3. We describe how we extend the basic wrapper design that was introduced in Section 3.1.

The design of our wrapper mechanism is motivated by insecurities in previous secure object sharing attempts. For example, one proposal [5] fails to protect the `__proto__` field of a shared function object, thereby opening up the frame to attack [2]. Another proposal presents self-protecting JavaScript wrappers [27], but we discovered several attacks by auditing their wrapper code examples. For example, their wrappers are vulnerable to at least two function prototype poisoning attacks, and their type forgery "solution" is vulnerable to a type forgery attack.

Complete Mediation. The basic recursive wrappers described in Section 3.1 wrap all of an object's properties and return values. We assign accessors or proxy functions to all fields on an object, including inherited ones; we also recursively wrap return values. Our wrappers also use reference equality to detect and restrict alternate access paths. Reconsider the example from Section 2.3 that tries to restrict access to `postMessage` by redefining `frame1.postMessage`. One reason this fails is because alternate access paths to `postMessage` exist. We can accomplish mediation correctly with a view. A view is made of the page document, and a policy is set on the view to restrict `frame1.postMessage`. The untrusted script receives this view. If the restricted script tries to access `frame1.postMessage` through an alternate path like `frame2.postMessage(frame1,m,url)`, our wrappers will detect reference equality between the two and apply the correct restrictions. In Caja, we use a simple dictionary lookup to check for reference equality; with our marshaling library, we associate object IDs with references.

Controlling Untrusted Parameters. The basic wrapper described in Section 3.1 is concerned with not letting references *out*, which we call "exporting". However, we must also be conscious of the effects of letting untrusted code *in*, which we call "importing". We import code when a view method accept parameters or permits a property reassignment. We do not want to pass privileged objects to callbacks on untrusted parameters. Parameters can also mount type forgery attacks. A reassigned property must not execute with a privileged object as its `this` parameter.

Our solution to protect views from imported code is to use dual *import* and *export* wrappers. An object defined by Alice will have an export wrapper: this is the basic wrapper based on membranes that we introduced in Section 3.1. Import wrappers surround any object introduced by another party (parameters and redefined properties). They are again recursive. Import wrappers prevent any of Alice's unwrapped objects from being passed into them, and imported methods execute with an export-wrapped version of its parent object as its local scope. To illustrate how import wrappers work, consider the following policy:

```
var x = {y: function () {}, secret: "secret"};
var c = makeView(x);
c.definePolicy([[x, {getters: {y: permit},
                     setters: {y: permit}}]]);
mallory.send(c.view); //x_w
```

Object `x` has two properties, one of which (`secret`) is not meant to be readable by the recipient of the wrapper. Now, the attempted attack:

```
x_w.y = function () { broadcast(this.secret); }; //Mallory
x.y(); //Alice
```

The view recipient redefines a property `y` to be a method that leaks the secret. If we did not have import wrappers, the new `y` function would broadcast the secret when the owner of `x` calls the method `y`. Consider how import wrappers prevent this attack:

1. `x_w` is an export wrapper for `x`, so the attempt to "set" field `y` of view `x_w` is subject to mediation.

2. The setter proxy sees that the function on the right hand side of the assignment is not wrapped. The setter applies an import wrapper to the untrusted function before assigning it to `x.y`.

3. When `x.y` is executed, the import wrapper's proxy function sets the context of `Function.apply` to be `x_w`. This means that the `this` object inside the function is export wrapped, and `secret` is protected.

By wrapping imported objects, we can prevent raw privileged objects from being passed to untrusted code and ensure that Alice's restrictions remain in place.

Wrapper and advice code must also be careful when handling shared objects because a malicious principal could redefine methods or properties that typically would have been inherited from `Object` or `Function`. Consider again the attack of adding a malicious and unexpected `toString` method to an object. We can enforce a policy that resets inherited methods to be the trusted versions of those methods.

Our trusted platform ensures that redefined methods have the correct global scope when they execute. With server-side script rewriting, a redefined method will still be subject to the original rewriting rules that define the global scope. With `postMessage`, code always executes in the frame that did the assignment, so it has the correct global scope.

Tamper-Proof Wrapper Methods. A wrapped object has one extra method beyond what is present on the original object: every wrapped object must have an unwrapping method for when an untrusted principal passes a protected argument to a protected method. Otherwise, the view owner loses pointer equality and faces restrictions on its own object. However, the unwrapping mechanism cannot be usable by an attacker. Our solution [29] is to communicate through a variable lexically scoped to the view controller. Calling `x_w.unwrap()` will set a variable in the view controller's scope to the original object and not return anything. Wrapper code can access the view controller's variable to retrieve that unwrapped object, but the attacker cannot. Invoking `x_w.unwrap` therefore leaks nothing to an attacker. Changing `x_w.unwrap` would only render the wrapper useless, since it would break its functionality. Additionally, it should be noted that deleting wrapper properties is not a concern for us. Server-side script rewriting tools can interpose on deletions, and we can prohibit cross-frame deletions.

Prototype Poisoning. A view wrapper's prototype can be accessed in two ways. First, `Object.prototype` is the root prototype for all of a given principal's objects. If an attacker could edit a view owner's `Object.prototype`, that change would be inherited by the view's wrapper. However, our trusted platforms provide initial isolation of principals, which protects the prototype chain. Server-side script rewriting provides this with rewriting rules. With the cross-frame system, all security-critical code runs in the sender's frame, which by the SOP is isolated from the recipient.

Access to the inheritance chain is also possible through the `__proto__` field of object instances, but we mediate this property the same way we mediate other properties.

We do not provide a formal proof of security; we leave this to future work since a usable formal semantics of JavaScript is still being formulated [20]. Alternately, a flow analysis could check that references do not leak. However, analysis has thus far been limited to small 50-250 line scripts [13] of subsets of JavaScript [14] and imprecisely model the DOM.

6. EVALUATION

All benchmarks are on a 2.4GHz Intel Core 2 Duo MacBook Pro with 2GB of RAM. Our measurements, unless otherwise noted, are Firefox 3.5.4 with JavaScript tracing optimizations enabled.

6.1 Prototype Status

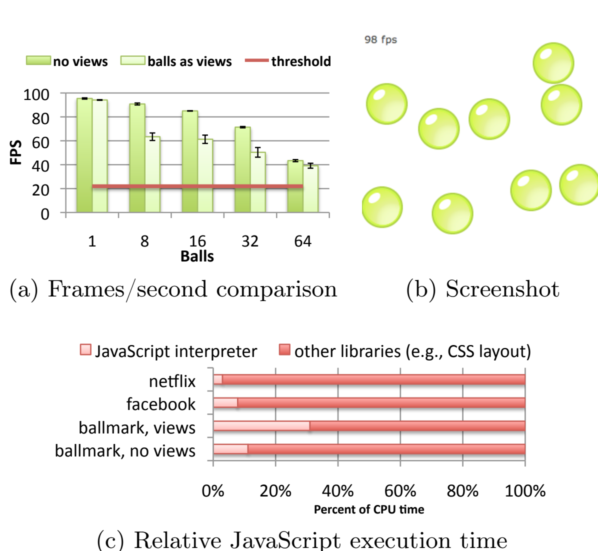
Our view creation, policy definition, and `postMessage` communication libraries conform to web standards. They have all been tested on Firefox, Safari, and Opera. We have synchronization bugs in our view creation library due to inconsistent introspection support by ECMAScript: we cannot detect new properties added to object property lists or prototype chains. Additionally, the lack of weak references in ECMAScript delays garbage collection.

Our library for sharing objects through `postMessage` is subject to the above challenges plus two additional problems related to the CPS transformer. The CPS transformer for our prototype is incomplete, as it ignores getters and setters. Until we fix the compiler, our proof-of-concept uses a library-level patch to attach semantically equivalent set and get methods. The transformer also might violate encapsulation; this is a problem when a principal both creates and receives views, since we need to transform our trusted view creation library into CPS style. We plan to either review the code and verify that encapsulation is maintained or manually CPS the library code to avoid the issue.

6.2 File size

File size is important for networked application performance: a small file loads faster. Our view creation and advice library is 445 lines of well-commented code. Our declarative policy system is 110 lines. The `postMessage` library is 334 lines, and it would not be necessary if browsers natively supported object marshaling.

Using standard JavaScript minifiers and ZIP file generators, our view creation library is 2.0KB. Including our policy library, it is 3.1KB. Adding the `postMessage` library increases the size to 7.4KB. To avoid the transformation cost of the CPS conversion, which is linear in application size, we can perform rewriting at runtime at a total size of 21.3KB. Our policies incur a constant, application-specific increase in code size. In contrast, weaving policies into the source increases application size from 1.5x to 5x [32, 25].



(a) Frames/second comparison (b) Screenshot

(c) Relative JavaScript execution time

Figure 5: Bubblemark benchmark with and without views. Views in all cases stay above the smooth animation threshold. Also, JavaScript execution time is dominated by other factors.

6.3 Speed Macrobenchmark

The Bubblemark benchmark for comparing user interface frameworks is an n-body animation of colliding balls [10] (Figure 5b). We compare a standard version of the UI benchmark against one in which every ball is wrapped in a view. All manipulations by the Ballmark JavaScript physics engine and the browser-provided layout engine go through views. Views in all cases stay above the smooth animation threshold (Figure 5a). Our `postMessage` library is not used.

JavaScript execution time is dominated by browser libraries like the layout engine. Using the Shark profiler, we sampled Safari 4.0.3's callstack at 20μs intervals over 2 seconds for the Bubblemark test and while loading post-login screens for`facebook.com` and `netflix.com` (Figure 5c). We found that the Bubblemark test is more JavaScript-intensive than those two sites, yet our view version still performs well enough to be unnoticeable (since we stay above the threshold). We expect that typical applications would be significantly less demanding than the Bubblemark test, but views still have sufficient performance in this worst case scenario.

6.4 Function Call Microbenchmarks

We examine four types of basic function calls (Figure 6), contrasting views with plain function calls and shallow ('lightweight') wrappers [27]. We implemented the shallow wrappers by setting accessors on object properties; as in their benchmarks [27], we simulate the cost of checking a policy by mutating a global variable in the accessor function. Note that the comparison is not apples-to-apples; our implementation does more work, such as wrapping return values and parameters. For each of the four function calls, we ran 20 trials consisting of 10,000 invocations of the call of interest:

- In the first test, we measure the overhead of making two DOM calls to set the font size of a paragraph. We found that the shallow wrappers impose an overhead of 53%, and views impose another 83% overhead (so 2.81x slower than the unwrapped version). This overhead is higher than would be expected because it is only JavaScript interpreter time. All modern browsers batch layout commands for later bulk processing; this means that the relevant (and expensive) browser library calls will occur sometime after the benchmark finishes.
- In the next test, a DOM write call is followed by a DOM read. By following a font size change with a read, we force the layout re-computation during the benchmark. We see interpreter time overhead is now only actually 6% for shallow wrappers and 22% overhead on deep ones.
- In the last two calls (a user-defined JavaScript function and a DOM call that does not use the layout engine), we see that shallow and deep wrappers have a cost linear in the number of calls, but do not significantly depend on the type of call.

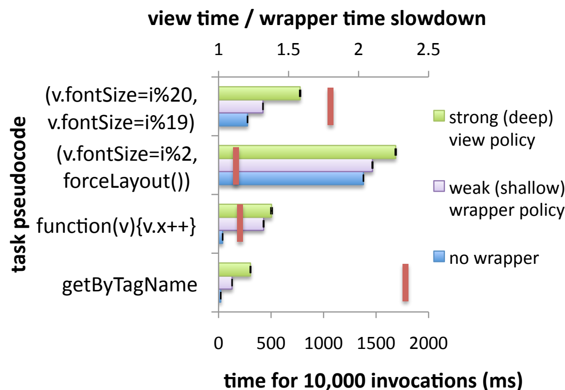


Figure 6: A comparison of unwrapped, shallowly wrapped, and deeply wrapped (view) performance on different microbenchmarks. The red lines show the relative slowdown of our views vs. the shallow wrappers.

Overall, we see that the overhead of enforcing a policy, when considering expensive JavaScript calls, is dominated by the calls themselves. While our system does have sophisticated runtime mechanisms, we only found the overhead relative to the lightweight approach to be 15% to 2.36x on impacted microbenchmarks. We do not perform microbenchmarks for `postMessage` object marshaling performance; authors of previous work provide this [2].

6.5 Memory

Strict use of ECMAScript delays garbage collection. The problem manifests itself when an object from Alice repeatedly passes through her view to Bob. As Bob repeatedly receives views of this object, he expects reference equality for them. Thus, the view controller maintains a map from raw objects to views. Consider when Alice does not reference the raw objects nor Bob the views. If Alice references the view controller, she also has an indirect reference to the map. To collect the map entries, the controller must also be gone. A fix enabled by a common ECMAScript extension is to encode a dictionary with weak values.

We measured the impact on memory. We ran two 20 minute trials at 99% CPU intensity, creating as many wrappers as it could to stress test the worst case scenario. The first trial created, used, and discarded views so that memory could be reclaimed. As expected, the browser cycled between 160MB and 200MB of RAM, signifying successful garbage collection. The second trial reused the same view, preventing reclamation of objects passing through it. We observed real memory use monotonically grew to 220MB, representing a 20MB increase despite full CPU load over a prolonged period. We conclude memory use is low and delayed collection is not inherent.

7. RELATED WORK

Membranes. Views are inspired by the *membrane* pattern, which controls an object by creating a recursive wrapper and tying it to an access control gate [22]. We extend the membrane pattern with an aspect system for sophisticated policies instead of a coarse access control gate. We also add pointer equality and many JavaScript security defenses to the canonical recursive membrane wrapper. A related mechanism was proposed for enforcing contracts that take developer-provided type annotations for functions and returns shared versions that enforce the signature [15]. Unfortunately, it is hard to write types for JavaScript programs and even a small developer mistake in such a signature may expose the entire system to attack. We conjecture that supporting policies like whitelists is less error-prone.

Aspects. One of our contributions is a notion of per-principal advice for multi-principal software. Prior aspect systems for web applications do not completely mediate access. One proposal [31] is not designed for an adversarial setting, and another [32] does not explain how it prevents the attack vectors identified in this paper. Two related proposals [27, 8] have vulnerabilities in their code samples. In general, these approaches suffer from incomplete mediation because their wrappers are installed directly on API properties and methods; this is a potential problem if the policy author fails to observe an unexpected path to a capability. Instead, we apply a single recursive wrapper to the whole API that checks reference equality on every operation.

Dantas et al. [6] also propose secure advice systems. Their security goal is to guarantee that malicious advice cannot interfere with certain program invariants. By requiring references to the raw object and the view in order to add advice – as opposed to global type-based pointcuts – we can assume advice has proper authority over the impacted objects and do not need to worry about their threat model. Instead, we concern ourselves with protecting advice from malicious view recipients.

Secure Browser Environments. Recent proposals like BEEP [18] and MashupOS [16] seek to tailor the granularity of the Same Origin Policy to the needs of a web site, e.g., to prevent unauthorized script execution or allow one-way DOM access. We are also interested in application-specific policies for sharing, but our sharing mechanism operates at a finer level. Other browser proposals have focused on improving isolation between principals [12]; we look at the next step of controlled sharing without violating isolation.

OMash [5] lets a frame define a public "interface" so that other principals may interact with it in a restricted fashion. This is similar in spirit to a view, but OMash limits value passing to primitives, whereas views support arbitrary objects. Our views could be used in conjunction with their framework to provide share objects over an interface. PostMash [2] encodes objects with object-to-string marshaling, and our `postMessage` library extends this idea with view advice to apply policies to marshaled sharing. Additionally, our view mechanism is broader and allows for the secure passing of actual references in scenarios where an object sharing communication channel is available.

Server-Side Script Rewriting. Our work on views originated as part of Google Caja [25]. Other server-side script rewriters (Facebook JavaScript [1] and Microsoft Web Sandbox [17]) have developed their own automated DOM wrapping systems since we began this research.

Lenses. Concurrent to our work, secure lenses were proposed [9] as a way to verify confidentiality of strings. Our focus is on supporting policies for sharing objects in web applications; how to embed lenses in languages used for applications is still unclear [11].

8. CONCLUSION

We propose *object views* as a user-level mechanism for fine-grained JavaScript object sharing. A view is an object proxy controlled by advice functions, which permits the expression of policies that govern access to the original object. Instead of sharing the actual object, a principal would share a view of the object. We build a policy system for developers to declaratively specify view restrictions; the policy system automatically generates advice functions from the declarative rules.

We present how views can be used in two settings: gadget aggregators with server-side script rewriting and cross-domain browser frames. Server-side script rewriters isolate gadgets from the rest of the page but need to provide restricted DOM access to gadgets; we propose views as a mechanism for partial DOM access. For cross-domain browser communication, we discuss how views can be exchanged over a `postMessage` object-to-string marshaling library. Marshaling objects over `postMessage` is not new [2]; we extend the idea with views to add advice-based policies.

Our security goal is to ensure that a view recipient cannot circumvent a view's restrictions or gain unauthorized access to additional references through a view. To this end, we implement views using a recursive wrapper that has been specialized to prevent JavaScript attacks. One of our contributions is an in-depth discussion of how to build JavaScript-safe wrappers.

Future work could examine further applications of views (e.g., partial DOM access for extensions), policy usability, security testing, and native browser support for views. We are particularly interested in examining the possibility of browser support for views.

9. REFERENCES

[1] FBJS - Facebok Developers Wiki. `http://wiki.developers.facebook.com/index.php/FBJS/`, 2008.

[2] A. Barth, C. Jackson, and W. Li. Attacks on JavaScript Mashup Communication. In *Web 2.0 Security and Privacy*, 2009.

[3] A. Barth, C. Jackson, C. Reis, and The Google Chrome Team. The Security Architecture of the

Chromium Browser. Technical report, Google Inc., 2008.

[4] A. Barth, J. Weinberger, and D. Song. Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense. In *18th USENIX Security Symposium*, 2009.

[5] S. Crites, F. Hsu, and H. Chen. OMash: Enabling Secure Web Mashups via Object Abstractions. In *15th ACM Conference on Computer and Communications Security*, pages 99–108, New York, NY, USA, 2008. ACM.

[6] D. S. Dantas and D. Walker. Harmless Advice. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 383–396, New York, NY, USA, 2006. ACM.

[7] O. Danvy. Back to Direct Style. In *4th European Symposium on Programming*, pages 130–150, London, UK, 1992. Springer-Verlag.

[8] U. Erlingsson, B. Livshits, and Y. Xie. End-to-End Web Application Security. In *11th USENIX Workshop on Hot Topics in Operating Systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[9] J. N. Foster, B. C. Pierce, and S. Zdancewic. Updatable security views. In *IEEE Computer Security Foundations Symposium (CSF), Port Jefferson, NY*, July 2009. To appear.

[10] A. Gavrilov. Balls animation test. `http://bubblemark/`, 2007.

[11] M. Greenberg and S. Krishnamurthi. Declarative, Composable Views, May 2007. Undergraduate Thesis.

[12] C. Grier, S. Tang, and S. T. King. Secure Web Browsing with the OP Web Browser. In *IEEE Symposium on Security and Privacy*, May 2008.

[13] S. Guarnieri and B. Livshits. Gatekeeper: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security Symposium*, Aug. 2009.

[14] A. Guha, S. Krishnamurthi, and T. Jim. Using Static Analysis for AJAX Intrusion Detection. In *18th International Conference on World Wide Web*, pages 561–570, New York, NY, USA, 2009. ACM.

[15] A. Guha, J. Matthews, R. B. Findler, and S. Krishnamurthi. Relationally-parametric polymorphic contracts. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 29–40, New York, NY, USA, 2007. ACM.

[16] J. Howell, C. Jackson, H. J. Wang, and X. Fan. MashupOS: Operating System Abstractions for Client Mashups. In *11th USENIX Workshop on Hot Topics in Operating Systems*, pages 1–7, Berkeley, CA, USA, 2007. USENIX Association.

[17] S. Isaacs and D. Manolescu. WebSandbox - Microsoft Live Labs. `http://websandbox.livelabs.com/`, 2009.

[18] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *16th International Conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.

[19] G. Kiczales. Aspect-Oriented Programming. *ACM Computer Survey*, page 154, 1996.

[20] S. Maffeis and A. Taly. Language-based Isolation of Untrusted Javascript. In *IEEE Computer Security Foundations Symposium*, 2009. See also: Dep. of Computing, Imperial College London, Technical Report DTR09-3, 2009.

[21] L. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for ajax applications. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2009*, 2009.

[22] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[23] M. S. Miller. Issue 1065: Security hole: Dangerous constructors still leaking. `http://code.google.com/p/google-caja/issues/detail?id=1065`, July 2009.

[24] M. S. Miller, J. E. Donnelley, and A. H. Karp. Delegating Responsibility in Digital Systems: Horton's "who done it?". In *2nd USENIX Workshop on Hot topics in Security*, pages 1–5, Berkeley, CA, USA, 2007. USENIX Association.

[25] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja - Safe Active Content in Sanitized JavaScript. `http://google-caja.googlecode.com/files/caja-spec-2007-10-11.pdf`, October 2007.

[26] N. Mix. Narrative JavaScript. `http://www.neilmix.com/narrativejs/doc/`.

[27] P. H. Phung, D. Sands, and A. Chudnov. Lightweight Self-Protecting JavaScript. In *4th International Symposium on Information, Computer, and Communications Security*, pages 47–60, New York, NY, USA, 2009. ACM.

[28] C. Reis, B. Bershad, S. D. Gribble, and H. M. Levy. Using Processes to Improve the Reliability of Browser-based Applications. Technical Report UW-CSE-2007-12-01, University of Washington, December 2007.

[29] M. Stieglar. E Fundamentals... with Donuts. `https://www.cypherpunks.to/erights/talks/efun/pingSealer.ppt`, 2004.

[30] 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 *18th USENIX Security Symposium*, Montreal, Canada, August 2009.

[31] H. Washizaki, A. Kubo, T. Mizumachi, K. Eguchi, Y. Fukazawa, N. Yoshioka, H. Kanuka, T. Kodaka, N. Sugimoto, Y. Nagai, and R. Yamamoto. AOJS: Aspect-Oriented Javascript Programming Framework for Web Development. In *8th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 31–36, New York, NY, USA, 2009. ACM.

[32] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript Instrumentation for Browser Security. In *34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 237–249, New York, NY, USA, 2007. ACM.