Object Views: Fine-Grained Sharing in Browsers

Leo A. Meyerovich[*], A. Porter Felt[†]
{lmeyerov, afelt}@eecs.berkeley.edu
University of California, Berkeley

Mark S. Miller
erights@google.com
Google Inc.

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. The security challenge is to fully mediate access to objects shared through a view and prevent privilege escalation. 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.

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. Developers can control the fine-grained behavior of object views with an aspect system that accepts programmatic policies.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; D.2.3 [**Software Engineering**]: Coding Tools and Techniques

General Terms

Security, Languages

Keywords

aspect, browser, ECMAScript, JavaScript, membrane, reference monitor, remote object, same origin policy, web security, wrapper

[*]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.

WWW 2010, April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

1. INTRODUCTION

Under current browser policies, sharing between web principals is all or nothing. Since sharing everything can lead to cross-site 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 objects between otherwise isolated principals [16, 4]. 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*, and a principal can share an object view instead of a plain object to limit the recipient's access to the shared object. 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, 21].

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. These systems must share heavily restricted DOM API access with gadgets and facilitate gadget-to-gadget communication.

- **Cross-origin sharing.** The `postMessage` browser primitive can be used to remotely share objects between frames by marshaling objects into strings. We present object views as a way to control and restrict remote object sharing in the browser.

Our primary contributions are 1) the abstraction of an *object view* for fine-grained object sharing patterns using an aspect system, and 2) 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. We begin by presenting attacks on secure object sharing within a single frame and show that a trusted platform is necessary to defend against some classes of attack. Our view mechanism is responsible for defending against the remaining attacks. We distinguish the responsibilities of the view mechanism from the trusted platform to make their relationship more explicit.

2.1 Web Security Model

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

The 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 origin. 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 origins.

2.2 Trusted Platforms

A *trusted platform* is responsible for providing encapsulation to separate principals from one another. Each *principal* is an instance of a script (or set of related scripts). We examine two trusted platforms:

Server-Side Script Rewriters. Gadget aggregators 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]. Rewritten scripts have separate "virtual" global objects. Several of the web's most visited sites (e.g., Facebook and the Yahoo! home page) use server-side script rewriting. Views are of interest to gadget aggregators because (a) they need to provide gadgets with restricted versions of DOM nodes and (b) they want to support object and DOM node sharing between gadgets. The former scenario is a one-way notion of security, and the latter is symmetric.

Browsers. The browser principals that we consider are frames from different origins. In this scenario, the browser provides isolation between the frames' principals as per the Same Origin Policy. Objects cannot be directly shared between cross-origin browser frames. Instead, the `postMessage` communication primitive can be used to *remotely share* objects by marshaling objects to strings. Using views, principals can control the way objects are remotely shared and prevent capability leaks.

2.3 Attacks

In our object sharing scenario, one principal shares an object by creating an object view with a policy and sending the view to another 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. We try to prevent privilege escalation by implementing a view as a wrapper around the shared object.

Consider an object sharing scenario where we have two web principals in the same frame, and one principal wants to share a restricted object with the other. The following example is an attempt to restrict an object from subsequent code in the page by wrapping the the object. The example wrapping technique is based on proposals by others [27, 20, 8]. The intended policy is to limit the content of a frame to URLs specified by a whitelist:

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

The method is reassigned early into loading a page to prevent access to the original `assign` 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 several categories of attacks:

1. **Incomplete mediation.** The `assign` function is still accessible in other ways. The DOM and other libraries often provide multiple ways to reference the same object, e.g., `frame2.location.assign.call(frame1, url)`.
2. **Incomplete policy.** `frame1.location.href` can also be used to change the location. This is different from incomplete mediation: with incomplete mediation, the same object can be referenced multiple ways; with an incomplete policy, the same capability can be accessed through multiple objects.
3. **Parameter type forgery.** 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.
4. **Root prototype poisoning.** All JavaScript objects inherit basic properties and methods from the `Object` prototype. In the above example, an attack can be mounted by adding a field to the Object prototype: `Object.prototype['http://z.com']`. As a result, the check on `wlist['http://z.com']` evaluates to true. Similar attacks can target the `Function` prototype and other globals.

Wrappers in a same-frame scenario are also subject to the following attack vectors:

5. **Prototype chain poisoning.** Prototype poisoning attacks do not require direct access to a root prototype. An attacker can traverse a wrapper's prototype chain to indirectly access a prototype: `obj.constructor.prototype` might point to a prototype that is shared with other objects.
6. **Untrusted parameter callbacks.** Raw objects should never be passed to untrusted code. For example, if a wrapper accepts an untrusted object as a parameter and then passes a raw object to a method on the untrusted object, then a raw object has been leaked.
7. **Callstack inspection.** ECMAScript 3 provides an exploitable form of stack inspection. For example, if a function is passed `{secret: 2}` for its first parameter and then the function calls another function, the called function can steal the secret with the following: `arguments.callee.caller.arguments[0].secret`
8. **Untamed JavaScript.** JavaScript can provide capabilities in surprising or excessive ways. For example, running `eval` on an untrusted string can be exploited to violate lexical scope.

Due to the current limitations of JavaScript, wrappers are unable to defend themselves against root prototype poisoning, attacks via the call stack, or untamed JavaScript features. We therefore must assume the presence of a trusted platform to remove these attack vectors by providing encapsulation. Our wrappers must defend against the remaining attacks: incomplete mediation, parameter type forgery, prototype chain poisoning, and untrusted parameter callbacks. Part of our contribution is exploring what security properties wrappers may provide and how they must rely upon an external platform. Further, although policy correctness is the responsibility of the developer, we provide mechanisms to help developers write better policies.

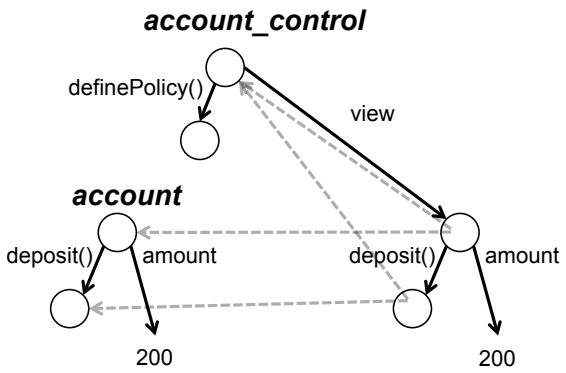
3. VIEWS FOR FINE-GRAINED SHARING

We present *object views*, an abstraction for securely sharing objects between principals. Views are shared 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.

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 an example 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 encodes a whitelist: Bob may read the `amount` property and invoke the `deposit` method,




```
var account = { deposit: function(v){ }, amount: 200 };
var account_control = makeView(account);
var permitGet = function (obj, prop) { return obj[prop]; };
account_control.definePolicy(account,
  {getters: {amount: permitGet, deposit: permitGet}});
var permitCall = function (f, t, a) { return f.apply(t, a); };
account_control.definePolicy(account.deposit,
  {funCall: permitCall});
Bob.send(account_control.view);
```

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

but he cannot redefine either nor any act upon any of Alice's other objects. Alice's access to the original `account` is not restricted, and she can make different views for the same object by making new control variables.

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 distinct object accessed through a view. A wrapper object is created by installing accessors (getters and setters) on the wrapper that correspond to all of the properties of the original object.

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 proxy. Reading and invoking a method are different operations: assigning `x = obj_v.deposit` will trigger a getter, whereas invoking a method `obj_v.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.

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 view of `bar`. Next, the getter that we have assigned to `bar_v.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`.

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

3.2 Aspect System

Views support fine-grained policies. To accomplish this, we implement a library-based 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 simple permit/deny policies, advice-based policies can significantly change behavior.

We implement whitelisting: a cross-principal action must be explicitly enabled. Whitelisting an action can be accomplished with the `permit` advice functions shown in Figure 1. More sophisticated advice can perform actions like translation or encryption. For example, consider Alice sending Bob a Spanish-translated version of her object:

```
function say_hi () { return "hello"; }
String.prototype.toSpanish = function () {
    return this == "hello" ? "hola" : "hola"; };
var c = makeView(say_hi);
c.definePolicy(say_hi, {funCall: function (f, ctx, args) {
      return f.apply(ctx, args).toSpanish(); }});
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 executable 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., `permitCall`) or custom policy functions. Giving a method a methCall rule will by default also set `read: permitGet` for that property; this is not necessary for function objects. Every rule specifies:

1. *Selector.* An XPATH expression selecting a set of descendent nodes to apply the rule to.

```
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;
```

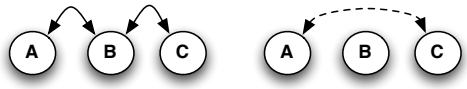
Figure 2: A policy that restricts a subtree to read-only if the root's class name includes `example`. Methods named `shake` may also be invoked.

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 specific rules apply to the same element, all of them will be applied as the union of the privileges.

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 would break expected invariants as has been explored with lenses [9]. 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.

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. Instead of specifying an `enabled:false` rule, the value `"obscured"` may be set. Our policy system then generates advice so that views of 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.



(a) normal linked list (b) shared view

Figure 3: A view hides 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 underlying 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

Several prominent gadget aggregators use server-side script rewriters to isolate untrusted scripts from the rest of the page. We focus on Google Caja [25], a server-side script rewriter which rewrites all gadgets into a subset of JavaScript and inserts runtime security checks. Caja supports two object sharing scenarios that could benefit from views: the gadget aggregator (*container*) needs to share a restricted version of the DOM API with gadgets, and gadgets might want to share objects with one another. Other gadget aggregators have similar object sharing needs.

For the container-gadget scenario, Caja currently uses a set of handwritten DOM wrappers known collectively as Domita. The Caja developers came up with a DOM taming policy and then manually implemented a different wrapper for each node type. 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. Automatically generated views from declarative policies might replace some handwritten wrappers. Additionally, our system could accommodate policies for other APIs (e.g., the OpenSocial API) that an aggregator might want to share with a gadget.

In the gadget-gadget sharing scenario, the two gadgets are mutually distrusting principals, given access to each other by the aggregator. Views could be used by either gadget to export mediated access to its own objects. These two scenarios compose. For example, gadget A migh provide gadget B read-only access to the DOM subtree that the aggregator has provided to gadget A. A's access to the real DOM is then limited to a view of the subtree according to the aggregator's policy, and B's access to the subtree is limited to a read-only view by gadget A's policy.

Caja provides security features that make it a suitable trusted platform: all prototypes are immutable, closures are truly encapsulated, global variables and the global object are inaccessible, the call-stack is inaccessible, and the whitelisted subset of JavaScript available to gadgets does not include JavaScript features that could escape Caja's security rules. We implemented a version of the view library to work with Caja. The library only uses JavaScript that meets the ECMAScript 3 specification; Caja provides the ability to install getters and setters on wrapper objects without using browser `getters` and `setters`. Caja also provides the ability to intercept the addition of a new property, which enables consistency between wrapper objects and raw objects. The current version of our Caja view library includes only a simple policy system, but we plan to integrate our full advice system with it.

4.2 Browser Frames

We propose views as a way to control remote object sharing between frames. Browsers do not provide a channel for reference passing across origin boundaries; instead, they provide an inter-frame string passing mechanism named `postMessage`. Objects can be remotely shared across this primitive by marshaling objects into strings and vice versa, as PostMash [2] demonstrates.

Although using `postMessage` means that no actual references are passed between windows, a marshaling library can still leak authority. Suppose that Alice implements a marshaling library that will carry out any operation on a remotely shared object. Alice remotely shares a form object

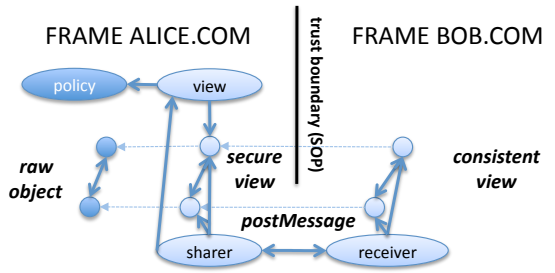


Figure 4: Remote object sharing with views.

`foo` with Bob, and then Bob asks the marshaling library for the value of `foo.parentNode.parentNode.parentNode.cookie`. This reveals the private value of the document cookie. If Alice remotely shared a view, the view could enforce a policy that only whitelists `foo`'s safe properties. Our contribution is how to apply policies to objects remotely shared over `postMessage` using views.

4.2.1 System Design

We present the use of views to safely share remote 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. The view creation library exists in sender Alice's frame because recipient Bob can manipulate any of the libraries in his frame. When Alice shares `tree_v` with recipient Bob, the sender library converts it to a string and sends it to Bob's frame using `postMessage`. The recipient library receives the message via a callback and turns the string into an object for Bob. When Bob later requests `tree_v.prop`, the marshaling 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 Alice's state needs to be protected from an adversarial recipient Bob but Bob desires no protection. 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.

Our implementation of the marshaling and view creation libraries for this embedding conforms to the ECMAScript 3 specification plus `postMessage`. We use field accessors in our original view design for clarity, but not in our actual implementation. Since our `postMessage` embedding requires a callback parameter for every action, sets and gets are more cleanly presented as functions.

Marshaling Library. Our marshaling library performs all of the necessary tasks related to converting objects. It encodes the basic JavaScript operations of calling functions, getting and setting fields, and throwing exceptions. Each message is a JSON object so we can rely on calls to standard browser support for JSON objects to prevent unexpected code execution. We also want to prevent our sender library from leaking additional information, so we refrain from using any global state in the sender library.

```
var z = x.y.z;
alert(z);
...
```
(a) synchronous

```
x_v.y(function (y) {
  y.z(function (z) {
    alert(z);
    ... }); });
```
(b) asynchronous

```
var z = x.y().z();
alert(z);
...
```
(c) implicitly yielding

```
var z_vP = x_v.y().z();
z_vP.when(
  function(z_v){alert(z_v);});
...
```
(d) promise-based

Figure 5: Shared object interaction options.

Functions and methods always execute in the frame that defined the function or method. 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.

Trusted Platform. The marshaling library and browser together serve as the trusted platform. The browser's SOP ensures general isolation between the two frames: they have separate prototype chains and sets of global objects. The separation of global objects means that potentially dangerous JavaScript features like `eval` are not an issue unless the view sender voluntarily chooses to make them available to the recipient. The marshaling library defends against leaks through stack inspection and root prototype poisoning. Our platform library checks objects before proxying them to ensure that transmitted objects are views of Alice's objects or proxies for Bob's. If a get, set, or call involves an object that is neither a view nor proxy, the sending library rejects the object. Attacks via stack inspection are therefore prevented: for example, the get `fn_v.caller.arguments` will fail because `fn_v.caller` is neither a view nor proxy object. We special case root prototypes for convenience. Requests for a root prototype object are rejected; instead, the receiving library substitutes the recipient frame's root prototypes. E.g., `fn_v.call(...)` uses the recipient frame's `Function.prototype.call`.

4.2.2 Asynchronous Shared Object Interactions

Typical JavaScript calls are *synchronous*, such as the example in Figure 5(a). However, the remote sharing of objects over `postMessage` introduces *asynchrony* because `postMessage` is an asynchronous[1] communication channel. If `x` in Figure 5(a) is changed to be a view of a remote object, the responses of the two "." calls become asynchronous. This introduces concerns about changing code structure and violating invariants naturally assumed in synchronous code. The challenges of asynchronous remote sharing are not introduced by our view abstraction; all cross-origin JavaScript work faces the same issues, and concurrent programming has long dealt with related challenges.

Our first concern is that using `postMessage` requires structurally inverting typical programs. Consider Figure 5(b), where we extend the PostMash [2] approach to remote object sharing by registering callbacks for every interaction. Not only must the developer rewrite the code that interacts directly with the shared object, she must also move subsequent code into a callback. This change affects the caller as well: the caller must supply a callback of where to continue upon completion of the segment. This pattern is known as *continuation passing style* (CPS) [6]. A client-side library can be used to automate the global transformation into CPS [26], enabling the interaction in Figure 5(c). In calls `y()` and `z()`, the proxy captures the continuation, registers a callback that continues it upon invocation, and yields the current thread of control. If ECMAScript natively supported continuations like the Rhino variant of JavaScript, then the rewriting step would be unnecessary.

A second concern is preserving invariants across calls. For example, control is never yielded in the original code in Figure 5(a), so we can locally reason that variable `alert` is bound to the intended function. However, yielding on the asynchronous call to `postMessage` introduces a race: a GUI event handler that breaks an invariant might be called while waiting for `z`, such as what `alert` is bound to. Figure 5(d) demonstrates a variant of promises [22], which can represent a delayed computation with a handle. Further asynchronous requests can be composed upon a handle, and the handles can be synchronously propagated. If a handle's actual value is needed, an asynchronously invoked callback may be registered to receive it. Depending on the implementation, rewriting might still be necessary (as in our example). Unlike callbacks, control is not entirely given to the receiver. Unlike the continuation-based approach, we avoid concerns about preemption.

5. VIEW SECURITY

A view recipient should only be able to access the shared object(s) according to the policy set by the view creator. We implement views with wrappers, and interpositioning occurs at points where reference leaks must be prevented. Our definition of wrapper safety is as follows:

A wrapper is secure if and only if: *A series of interactions with a wrapper can return only a wrapped value, a primitive value, or an unwrapped value previously passed in to the wrapper.*

To achieve wrapper safety, the attacks described in Section 2.3 must be addressed. We rely on trusted platforms (Section 4) to initially isolate principals and provide security defenses against stack inspection, root prototype poisoning, and JavaScript that can escape encapsulation. Our wrapper mechanism, assuming the above, must provide complete mediation, deal with parameter type forgery and untrusted parameter callbacks, and prevent prototype chain poisoning. By carefully considering these attack vectors, we aim to avoid past mistakes; for example, an audit of a proposal for self-protecting JavaScript wrappers [27] reveals at least two prototype poisoning attacks and a parameter type forgery "solution" that is still vulnerable to a type forgery attack.

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 use reference equality to detect and restrict alternate access paths. Reconsider the example from Section 2.3 that tries to restrict access to `assign` by redefining `frame1.assign`. One reason this fails is because alternate access paths to `assign` exist. We can accomplish mediation correctly with a view. We look up a policy based on the reference equality of the object under consideration,

[1]Internet Explorer non-compliantly implements `postMessage` synchronously, avoiding the issues in this section.

and `frame1.assign` and `frame2.assign` refer to the same closure. Consequently, the same policy applies regardless of the aspect path. In Caja, we use a dictionary lookup to check for reference equality; with our marshaling library, we associate object IDs with references.

Untrusted Parameters. The basic wrapper described in Section 3.1 is concerned with not letting raw 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 accepts parameters or permits a property reassignment. Imported code raises three safety issues: we should not pass privileged objects to callbacks on untrusted parameters or methods that the view recipient has redefined; we must defend against parameter type forgery attacks; and a property that the view recipient has reassigned should not execute with a privileged object as its `this` parameter.

A solution to protect views from imported code is to use dual *import* and *export* wrappers [29]. An object defined by Alice will have an export wrapper: this is the basic wrapper based on membranes. Import wrappers surround any object introduced by another party (parameters and redefined properties). They are identical to export wrappers, except that they add two extra security checks. 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. Consider the following policy:

```
var x = {y: function () {}, secret: "secret"};
var c = makeView(x);
var permitSet = function (o, p, rhs) { o[p] = rhs; };
c.definePolicy(x, {getters: {y: permitGet},
                   setters: {y: permitSet}}]);
mallory.send(c.view); // gives Mallory the view of x
```

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

```
// x_v is Mallory's view of x
x_v.y = function (o) { broadcast(o.secret); };

// Alice calls y on the original object
c.definePolicy(x.y, {funCall: permitCall});
x.y(x);
```

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. The import wrapper on `x.y` discovers that its argument (Alice's secret) is protected state. The argument will thus be wrapped for export, and `secret` will not be a property of the export wrapper.

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 from Section 2.3 of adding a malicious `toString` method to an object. The malicious version of `toString` provides different answers at different times. We can enforce a policy that sets inherited methods on import wrappers to be the trusted versions of those methods.

Prototype Chain Prototype Poisoning. For security, the prototype chain of a view must only contain wrapped objects. For consistency, elements along the prototype chain of a view should proxy to their unwrapped counterparts. The psuedo-code to wrap object `o` (assuming it does not directly inherit from a root prototype) in ECMAScript 3 is:

```
var cnstrctr_v = function () {};
cnstrctr_v.prototype = wrap(o.constructor.prototype);
var o_v = new cnstrctr_v();
```

The prototype of `o_v` is a wrapped object so, inductively, `o_v`'s inheritance chain should be of wrapped objects. Attacks like deleting fields of a view to expose underlying prototype or constructor values are made meaningless as the wrapper's prototype chain consists of wrappers. Our approach fails when we reach the bottom of the prototype chain. The final wrapper should have a prototype of `null`, but running

```
cnstrctr_v.prototype = null;
o_v = new cnstrnctr_v();
```

gives `o_v` a prototype of `Object.prototype`. We rely on the trusted platform to protect root prototypes.

Modern JavaScript variants typically expose the prototype property as the readable and writeable field `__proto__`, in which case we do not need `cnstrctr_v` as we can simply write `o_v.__proto__ = null`. Wrapping the prototype chain of functions currently requires the non-standard field `__proto__` as we cannot set the function constructor prototype. Without the `__proto__` field, we lose some consistency. We can add inherited properties and fields directly to a function's wrapper, but the actual prototype chain is lost.

Privileged Unwrapping. An object wrapper has one extra method beyond what is present on the original object: every object wrapper 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 exploitable by an attacker. One solution [30] 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 or deleting `x_w.unwrap` would only render the wrapper useless, since it would break its functionality. Import wrappers have unwrap methods just like export functions, but an attacker should not be able to invoke an import wrapper's unwrap method because the attacker should never come into contact with an import wrapper.

6. EVALUATION

All benchmarks are on a 2.4GHz Intel Core 2 Duo MacBook Pro with 2GB of RAM. We measured the performance of our view creation library in Firefox 3.5.4 with JavaScript tracing optimizations enabled. We tested a version of our code that uses JavaScript accessors.

6.1 File size

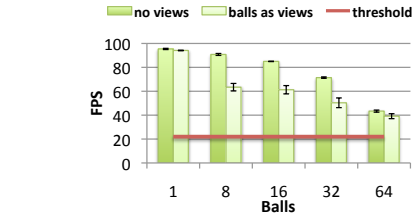
File size is important for networked application performance: a small file loads faster. Our view creation and advice library is 445 lines of well-commented code. Our declarative policy system is 110 lines. The `postMessage` library is 334 lines, and it would not be necessary if browsers natively supported 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. Our policies incur a constant, application-specific increase in code size. In contrast, weaving policies into the source increases application size from 1.15x to 1.65x [20].

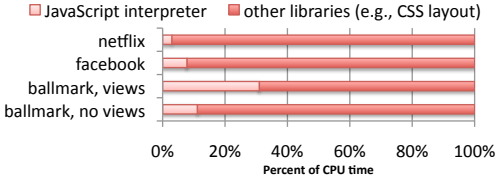
6.2 Speed Macrobenchmark

The Bubblemark benchmark for comparing user interface frameworks is an n-body animation of colliding balls [10]. 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 6(a)). 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 6(b)). We found that the Bubblemark test is more JavaScript-intensive than those two sites, yet the view version still performs well enough to 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.



(a) Frames/second comparison



(b) Relative JavaScript execution time

Figure 6: 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 Function Call Microbenchmarks

We compare the performance of views with plain function calls and shallow (non-recursive) wrappers [27] on four types of basic function calls (Figure 7). 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. Views do more work than shallow wrappers, 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 expensive layout 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.

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].

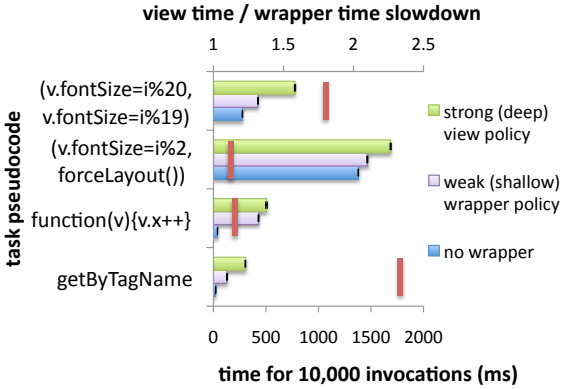


Figure 7: A comparison of unwrapped objects, shallowly wrappers, and deep wrappers on different microbenchmarks. The red lines show the relative slowdown of our views vs. the shallow wrappers.

6.4 Memory

Use of ECMAScript 3 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. However, consider when Alice does not reference the raw objects nor Bob the views. If Alice references the view controller, she still has an indirect reference to the map. To collect the map entries, the controller must also be gone. ECMAScript Harmony includes a proposal for ephemeral weak tables, which would solve this problem [7].

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.

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 consider JavaScript-specific security. 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 two [20, 8] do not explain how they prevent the attack vectors identified in this paper. Two related proposals [27, 8] have vulnerabilities in their code samples. Most of these approaches suffer from incomplete mediation because their wrappers are installed indirectly as 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 for policies based on object identity.

Dantas et al. [5] 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 [4] 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. Advice functions permit 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 gain unauthorized access to unrestricted 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 a 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.

9. ACKNOWLEDGEMENTS

We would like to thank David Wagner, David Molnar, and Benjamin Hindman for their insightful comments. We would also like to thank the Google Caja team for their help with the Caja library implementation.

10. 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] 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.

[5] 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.

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

[7] ECMAScript Wiki. weak references. `http://wiki.ecmascript.org/doku.php?id=strawman:weak_references`.

[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] H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov. JavaScript Instrumentation in Practice. In *ASPLAS*, 2008.

[21] 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.

[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] F. Spiessens. *Patterns of Safe Collaboration*. PhD thesis, Université catholique de Louvain, 2007.

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

[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.