Jigsaw: Efficient, Low-effort Mashup Isolation

James Mickens
Microsoft Research
mickens@microsoft.com

Matthew Finifter
University of California, Berkeley
finifter@cs.berkeley.edu

Abstract

A web application often includes content from a variety of origins. Securing such a mashup application is challenging because origins often distrust each other and wish to expose narrow interfaces to their private code and data. Jigsaw is a new framework for isolating these mashup components. Jigsaw is an extension of the JavaScript language that can be run inside standard browsers using a Jigsaw-to-JavaScript compiler. Unlike prior isolation schemes that require developers to specify complex, error-prone policies, Jigsaw leverages the well-understood public/private keywords from traditional object-oriented languages, making it easy for a domain to tag internal data as externally visible. Jigsaw provides strong iframe-like isolation, but unlike previous approaches that use actual iframes as isolation containers, Jigsaw allows mutually distrusting code to run inside the same frame; this allows scripts to share state using synchronous method calls instead of asynchronous message passing. Jigsaw also introduces a novel encapsulation mechanism called surrogates. Surrogates allow domains to safely exchange objects by reference instead of by value. This improves sharing efficiency by eliminating cross-origin marshaling overhead.

1 Introduction

Unlike traditional desktop applications, web applications are often *mashups*: applications that contain code from different principals. These principals often have asymmetrical trust relationships with each other. For example, a page that generates localized news may receive data from a news feed component and a map component; the integrating page may want to isolate both components from each other, and present them with an extremely narrow interface to the integrator's state. As another example, a social networking page might embed a third-party application and an advertisement. The integrating page may expose no interface to the advertisement. However, if the developer of the third-party application has signed a terms-of-use agreement, the integrating page may expose a relatively permissive interface to its local state.

Given the wide range of trust relationships that exist between web principals, it is challenging for developers to create secure mashups. Principals often want to share with each another, but in explicit and controlled ways. Unfortunately, JavaScript (the most popular client-side scripting language) was not designed with mashup security in mind. JavaScript is an extremely permissive language with powerful reflection abilities but only crude support for encapsulation.

1.1 Previous Approaches

Given the increasing popularity of web services (and the deficiencies of JavaScript's isolation mechanisms), a variety of mashup isolation frameworks have emerged from academia and industry. Unfortunately, these frameworks are overly complex and present developers with an unwieldy programming model. Many of these approaches [2, 11] force developers to use asynchronous, pass-by-value channels for cross-principal communication. Asynchronous control flows can be difficult for developers to write and understand, and automated tools that convert synchronous control flows into asynchronous ones can introduce subtle data races (§2.1). Additionally, marshaling data over pass-by-value channels like `postMessage()` can introduce high serialization overheads (§4.2).

Prior mashup frameworks also present developers with complex, overly expansive APIs for policy specification. For example, object views [11] require developers to define policy code that runs during each property access on a shared object. Understanding how these filters compose across large object graphs can be difficult. Similarly, ConScript [12] policy files consist of arbitrary JavaScript code. This allows Conscript policies to be

extremely general, but as we demonstrate, such expressive power is often unnecessary. In many cases, secure mashups require only two policy primitives: a simple yes-no mechanism for marking JavaScript state as externally sharable, and a simple grammar based on CSS and regular expressions that constrains how untrusted code can access browser resources like persistent storage and the visual display.

1.2 Our Solution: Jigsaw

In this paper, we introduce Jigsaw, a new framework for mashup isolation. Jigsaw allows JavaScript code from mutually distrusting origins to selectively expose private state. Jigsaw's design was driven by four goals:

Isolation by default: In Jigsaw, an integrating script includes guest code which may hail from a different origin. The integrator has access to browser resources, and the integrator can provide its guests with access to some portion of those resources. However, by default, a guest cannot generate network traffic, update the visual display, receive GUI events, or access local storage. Similarly, each principal's JavaScript namespace is hidden from external code by default, and can be accessed only via public interfaces that are explicitly defined by the owning principal.

Efficient, synchronous sharing: Jigsaw eschews asynchronous, pass-by-value sharing in favor of synchronous, pass-by-reference sharing. Inspired by traditional object-oriented languages like Java and C++, Jigsaw code uses the `public` and `private` keywords to indicate which data can be accessed by external domains. When an object is shared outside its local domain, Jigsaw automatically wraps the object in a *surrogate* object that enforces public/private semantics. By inspecting surrogates as they cross isolation boundaries, Jigsaw can "unwrap" surrogates when they return to their home domain, ensuring that each domain accesses the raw version of a locally created object. Jigsaw also ensures that only one surrogate is created for each raw object. This guarantees that reference-comparison $==$ operations work as expected for surrogates. Using surrogates, Jigsaw can place mutually distrusting principals inside the same iframe while providing iframe-style isolation and pass-by-reference semantics. Since principals reside within the same iframe, no `postMessage()` barrier must be crossed, which allows for true synchronous interfaces.

Simplicity: Using deny-by-default policies for browser resources like network access, and using the `public` and `private` modifiers to govern access to JavaScript namespaces, Jigsaw can express many popular types of mashups—most require only a few lines of policy code and the explicit definition of a few public interface methods. In designing Jigsaw, we consciously avoided more complex isolation schemes like information flow control, object views [11], and ConScript [12]. While these schemes are more expressive than Jigsaw, their interfaces are unnecessarily complex for many of the mashup patterns that are found in the wild.

Fail-safe legacy code: In most cases, regular JavaScript code that has not been adapted for Jigsaw will work as expected when used within a single domain. In all cases, unmodified legacy code will fail safely (i.e., leak no data) when accessed by external domains. Jigsaw prohibits some JavaScript features like dynamic prototype manipulation, but these features are rarely used by benevolent programs (and are potentially exploitable by attackers) [1, 11]. Jigsaw makes few changes to the core JavaScript language, and we believe that these changes will be understandable by the average programmer, since the changes make JavaScript's object model behave more like that of a traditional, class-based OO language like C#. Jigsaw preserves many of the language features that make JavaScript an easy-to-use scripting language. For example, Jigsaw preserves closures, first-class function objects, object literals, an event-driven programming model, and pass-by-reference semantics for all objects, not just those that are shared within the same isolation domain.

2 Design

In Jigsaw, *domains* are entities that provide web content. In the context of the same-origin policy, a Jigsaw domain corresponds to an origin, and we use the terms "domain" and "origin" interchangeably. A *principal* is an instance of web content provided by a particular domain. A principal may contain HTML, CSS, and JavaScript. Note that some principals might contain only JavaScript, e.g., a cryptographic library might only define JavaScript functions to be invoked by external parties.

A user visits top-level web sites; each of these sites can be an *integrator* principal. An integrator may include another principal P_i by explicitly downloading content from P_i's origin. In turn, P_i may include another principal P_j. Figure 1 depicts the relationship between the user and this hierarchy of client-side principals. When there is an edge from P_i to P_j, we refer to P_i as the *including principal* or *parent*, and P_j as the *included principal* or *child*.

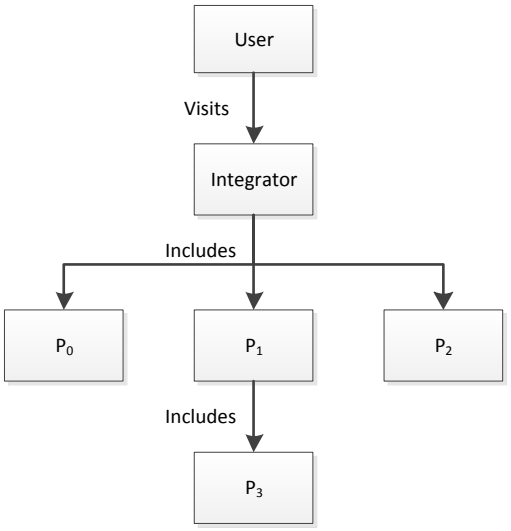

Figure 1: An example of a principal hierarchy. The user visits an integrator site that includes other principals. Each of those principals may include additional principals.

2.1 Boxes

Jigsaw places each principal in an isolation container that we call a *box*. Each box is associated with the following resources:

- A *JavaScript namespace* containing application-defined objects and functions.
- A *DOM tree*, which is a browser-defined data structure representing the HTML and CSS content belonging to the principal.
- An *event loop*, which captures mouse and keyboard activity intended for that box.
- A rectangular *visual region* with a width, a height, a location within the larger browser viewport, and a z-axis value.
- A *network connection*, which allows the principal to issue HTTP fetches for data.
- A *local storage area*, which stores cookies and implements the DOM storage abstraction.

In some ways, a Jigsaw box resembles a traditional iframe. Both provide a principal with a local JavaScript namespace, DOM tree, event loop, and visual field. However, Jigsaw boxes differ from iframes in three important ways.

First, if two iframes share the same origin, they can directly access each other's JavaScript namespaces via frame references like `window.parent` and `window.parent.frames`. A Jigsaw box does not allow such unfettered cross-principal access. By default, two boxes have no way to communicate with each other, even if they belong to the same origin. This enables fault isolation and privilege separation for different principals that originate from the same domain. To allow cross-box communication, each principal must explicitly define public functions on a *principal object*. By exchanging principal objects with each other, boxes define the set of external domains with which they communicate, and the operations that these domains may invoke on private state.

```
//Download a script and place it inside a
//new box. The return value of the script
//is its principal object.
var p = Jigsaw.createBox('http://x.com/box.js',
                  {network: /(x|y)\.com/,
                   storage: true, dom: null});

//Call a public method defined by the box.
//The pass-by-reference of localData and
//result is made safe by the use of
//surrogates.
var result = p.f(localData);
```

Figure 2: An integrator creating and interacting with a box. The box can exchange network traffic with servers from x.com and y.com; it can also access its origin's DOM storage, but it cannot access the integrator's DOM. Surrogates are explained in Section 2.6.2.

A second difference between iframes and Jigsaw boxes is that boxes use nesting relationships to more tightly constrain the resources of children. A page's top-most Jigsaw box is given a full visual field that is equivalent to the entire browser viewport. The root box is also given the maximal network permissions allowed by the same-origin policy. By default, descendant boxes lack access to non-computational browser resources. For example, child boxes cannot issue network requests, and they cannot access the visual field (and thus they cannot receive GUI events from the user). A parent box may delegate a region of its visual field to a child. Similarly, a parent can grant a child box a portion of its network permissions. In both cases, parent-child delegation has monotonically increasing strictness, i.e., a parent can never give a child a larger visual field or more networking permissions than the parent has. Figure 2 shows an example of how an integrator creates a new box. Note that DOM storage ACLs are unique because the browser gives each origin a local storage area. Thus, an integrator can grant a child access to the child domain's local storage, or completely prohibit such accesses. However, the integrator cannot directly expose its own local storage to a child domain (although it can define a method on its principal object that mediates access to its DOM storage).

The final difference between iframes and boxes involves communication channels. Applications in different iframes communicate with the asynchronous

```
var x = 0;  //Global variable

function increment(){
    x++;
}

function f(){
    alert(x);
    increment();
    alert(x);
}

//Create two tasks that fire every 100 ms.
setInterval(f, 100);
setInterval(f, 100);
```

Figure 3: In regular JavaScript, all code is given the illusion of single-threaded execution. Thus, in this example, only one version of `f()` can be executing at any given time. This means that the `alert()` statements from a particular instance of `f()` will always output two consecutive numbers.

```
function f(){
    alert(x);
    increment->(); //This version of f() now
                   //relinquishes the CPU!
    alert(x);
}
```

Figure 4: If increment() is asynchronous and replaced with a pseudo-synchronous continuation, a particular invocation of `f()` may be swapped off of the processor without executing atomically, allowing the other `f()` to execute fully, and causing the first one to output 0 and then 2. This would be impossible if the `->` continuation operator provided true synchronous semantics.

`postMessage()` call. `postMessage()` can only transmit immutable strings; thus, passing large objects across this channel can incur significant marshaling overhead, and explicit message passing is needed to keep mirrored data structures synchronized between iframes. The asynchronous nature of `postMessage()` also makes it difficult to provide true synchronous RPC semantics. Although tools exist to convert asynchronous function calls into continuation-passing style (CPS) [18], CPS can introduce data races that do not exist when function calls are truly synchronous (see Figures 3 and 4). Ensuring that such races do not exist requires the programmer to explicitly reason about synchrony and use application-level coordination mechanisms like locks.

Besides the performance and correctness challenges, asynchronous `iframe/postMessage` architectures are often ill-suited for many mashup designs. For example, consider pure computational libraries. If an integrator has some data that it wants to process using N calls to a cryptographic library or an image manipulation library, it is cumbersome for the integrator to set up a chain of asynchronous callbacks that executes the $(i+1)^{th}$ operation when the i^{th} operation has completed. A synchronous programming model is much more natural.

There are also event-driven mashups that are ill-suited for the asynchronous `iframe/postMessage` model. For example, suppose that an integrator uses an external library to sanitize AJAX data as it streams in. As chunks of data arrive, the browser fires the `XMLHttpRequest` callback multiple times. During each invocation, the integrator must pass the new AJAX data to the sanitization library. However, if the sanitizer lives in a separate `iframe`, the integrator cannot receive the sanitized data immediately—instead, the integrator must buffer data and wait for the sanitizer to asynchronously return the scrubbed results. This introduces two sources of asynchrony (the `XMLHttpRequest` handler and the sanitizer callback) when only one should be necessary (the `XMLHttpRequest` handler).

A Jigsaw application can have multiple boxes, but all of the boxes live in the same frame. As shown in Figure 2, this means that data can be passed synchronously and by reference. As described later, Jigsaw code uses the `public` and `private` modifiers to indicate which methods and variables are accessible to external domains. The Jigsaw runtime uses these modifiers to validate cross-domain operations (§2.6.2).

A single Jigsaw application may contain multiple principals that originate from the same domain. Jigsaw places each of these principals in a separate box. These principals interact with each other using the same public interfaces that are used by principals from different domains.

2.2 Principal Objects

A *principal object* defines the public functions and variables that a domain exposes to untrusted code; to communicate with an external domain, a box must possess a reference to that domain's principal object. A principal can access its parent's principal object by calling `Jigsaw.getParentPrincipal()`. Similarly, the `Jigsaw.principals` array contains principal objects for all of a box's immediate children. `Jigsaw.getRootOriginPrincipal()` returns the principal object belonging to the highest-level box from the caller's origin. This ancestor can act as a coordination point for all principals from that domain, e.g., if the principals want to synchronize their writes to DOM storage. Principal objects can be passed between boxes like any other object.

2.3 The DOM Tree

Each Jigsaw box can potentially contain a DOM tree and an associated visual field. If the box's parent did not del-

egate a visual field, Jigsaw sets the box's DOM tree to null, and prevents the child principal from adding DOM nodes to it. Otherwise, the child may update its visual field and receive GUI events for that field by modifying the DOM tree in the standard way. Jigsaw ensures that the visual updates respect the constraints defined by the parent.

A visual field consists of a width, a height, a location within the parent's visual field, and a z-order. Parents specify these parameters using CSS-style syntax. Thus, a child can have a static visual geometry, or one that flows in dynamic ways, e.g., to occupy a percentage of the parent's visual field, regardless of how the parent is resized.

Visual parameters are associated with each principal object (e.g., `principal.height`). A parent can dynamically change a child's visual field by writing to these fields. If a child wants to change its visual field, it can also try to write to these fields. However, the changes must be validated by the parent. A child write to a visual field parameter fires a special Jigsaw event in the parent called `childVisualFieldRequest`. If the parent has registered a handler for this event, the handler inspects the changes requested by the child. If the parent approves the change, the handler returns true, otherwise it returns false. Jigsaw will only implement the change if the parent has defined such a handler, the handler returns true, and the change would not place the child's visual field outside the one owned by the parent.

Similar to the Gazelle browser [24], Jigsaw requires visually overlapping boxes to be opaque with respect to each other. In other words, boxes cannot request transparent blending of their overlapping visual region—the box with the higher z-order occludes all others in the stack. This prevents a malicious box from making itself transparent, creating a child box containing a victim page, and then collecting GUI events that the user intended to send to the victim box.

Even with these protections, a principal must still trust its ancestors in the principal hierarchy. This is because a malicious parent can virtualize a child's runtime (§3) in subversive ways, or not create a child at all. Jigsaw can only guarantee that parents are protected from descendants, and that sibling principal hierarchies are protected from each other if the shared parent is non-malicious.

2.4 Network Access

A principal uses HTTP requests to communicate with remote servers. The principal's parent controls the set of servers that are actually reachable. Like visual field permissions, network privileges nest in a monotonically restrictive way. The most expansive privilege is "*", which means that a principal can fetch any resource that is allowed by the same-origin policy. Parents can also restrict children to a limited set of accessible domains. Parents can specify a group of related domains using a straightforward wildcard syntax, e.g., `*.foo.com` or `cache.*.bar.com`. As a syntactic shortcut, a parent can specify the "self" domain to indicate that the child can communicate with servers from the child's origin. Similarly, the "parent" token resolves to the parent's origin.

2.5 Local Storage

In HTML5, the DOM storage abstraction allows each origin to maintain a client-side key/value database. Each database can be accessed only by JavaScript code from the associated origin. Jigsaw partitions DOM storage in the same way. If principals from different origins want to exchange data from their respective DOM storage areas, they must do so via public interface methods.

2.6 The JavaScript Namespace

In traditional JavaScript, objects are dictionaries that map property strings to values. Using JavaScript's extremely permissive reflection interface, a program can dynamically enumerate an object's properties and read, write, or delete those properties. Unlike standard class-based languages like Java and C#, JavaScript uses prototype-based inheritance. A prototype object is an exemplar which defines the property names and default property values for other instances of that object. By setting an object's `__proto__` property to the exemplar, the object becomes an instance of the prototype's class. By setting the `__proto__` fields of prototype objects, one creates inheritance hierarchies. By default, an object's property list is dynamic, so an instance of a particular prototype can dynamically gain additional properties that are not defined by the prototype. An object's `__proto__` field is just another property, meaning that an object's class can dynamically change as well.

These default reflection semantics are obviously unsuited for cross-domain encapsulation. JavaScript does allow a limited form of data hiding using closures, which are functions that can access a hidden, non-reflectable namespace. Unfortunately, closures are an imperfect substrate for cross-domain sharing. They cannot be shared across iframes, and within an iframe, each closure has unfettered access to the DOM tree, event loop, and network resources that belong to the enclosing frame. Furthermore, closures are clumsy to program and maintain, since the hidden closure variables are implicitly obscured via lexical scoping instead of explicitly marked via a special keyword. Jigsaw provides simpler, stronger encapsulation using boxes and the `public`/ `private` keywords.

```
function Ctor(x, y){
    public this.x = -1;
    private this.y = 42;
    this.z = 100;   //By default, a new
                    //field is private.
}

public Ctor.prototype.prop1 = "hi";
private Ctor.prototype.prop2 = "bye";
Ctor.prototype.prop3 = "aloha"; //Private
                                //by default

obj = {};
public obj.a = 0;
private obj.b = 1;
obj.c = 2;  //Private by default.
```

Figure 5: Jigsaw code example (private-by-default property visibility).

```
function Ctor(){}
private Ctor.prototype.x = 0;

obj = new Constructor();
obj.x = 42; //Succeeds: Private properties
            //visible within the creating box.
public obj.x; //Fails: can't override modifier
              //specified by prototype
```

Figure 6: Jigsaw code example (visibility modifiers flow from prototypes to instances).

2.6.1 Visibility modifiers

The `public` and `private` keywords allow a principal to define which object properties are visible when an object is shared across boxes. For example, if a principal from domain X creates the following object ...

```
var obj = {public x:  "foo",
           private y:  "bar"};
```

... and passes it to domain Y, Y can access `obj.x` but not `obj.y`. Note that "access" means the ability to read, write, and delete a property.

The `public`/`private` keywords can be used anywhere a variable is declared. If a variable is declared and no visibility modifier is specified, *it is private by default*, as shown in Figure 5. When code from domain X enumerates the properties of an object from domain Y, private fields do not appear in the enumeration. Within Y, private fields do show up in the enumeration.

As shown in Figure 6, public and private modifiers "flow downward" from prototypes to instances, overriding any attempts by instance objects to reset the modifiers. JavaScript allows object properties to be declared at arbitrary moments, so during program execution, when Jigsaw encounters a `public` or `private` statement, it must dynamically check whether the statement satisfies the visibility settings for the relevant prototype object.

When Jigsaw passes objects between boxes using surrogates (§2.6.2), it never exposes object prototypes or constructor functions. This prevents a wide class of exploits called prototype poisoning [1, 11] in which an attacker dynamically modifies the inheritance chain for an object and subverts the object's intended implementation.

2.6.2 Surrogate Objects

Jigsaw uses surrogate objects to enforce public/private semantics. When an object `obj` is passed between boxes, e.g., when box X invokes a function on Y's principal object and passes a local object `obj` as an argument, Jigsaw wraps `obj` in a surrogate and passes that surrogate, not the original object, to the destination domain Y. To create the surrogate, Jigsaw first creates an initially empty object. Then, for each public property belonging to `obj`, Jigsaw adds a getter/setter pair for a corresponding property on the surrogate object. Getter/setters are a JavaScript feature that allow an object to interpose on reads and writes to a property. For `obj`'s surrogate, the getter for property `p` returns `createSurrogate(obj.p)`. The setter for property `p` executes `obj.p = createSurrogate(newVal);`.

The `createSurrogate()` function has several important features. First, `createSurrogate()` associates at most one surrogate for each "raw" object. Thus, calling `createSurrogate(obj)` multiple times on the same object will always return the same surrogate object. This ensures that the reference-compare $==$ operator has the expected semantics for surrogates. For example, if a box is passed two surrogates from two different domains, and those surrogates refer to the same backing object, then the surrogates will reference-compare as equal.

Another important property of `createSurrogate()` is that it does not always return a surrogate object. For immutable, pass-by-value primitive properties like numbers, `createSurrogate()` returns the primitive value. More interestingly, if a surrogate is being passed to its originating box, `createSurrogate()` returns the backing object. In the previous example, this means that if Y passes `obj`'s surrogate back to X, Jigsaw will "unwrap" the surrogate and hand the raw object back to X. This convenient feature also helps $==$ to work as expected, since boxes do not need to worry about receiving a surrogate for a local object that lacks reference equality with that local object.

A final property of `createSurrogate()` is that surrogate getter/setters invoke it lazily—if a surrogate property is never accessed by external boxes, the sur-

rogate will never call `createSurrogate()` for that property. As we show in Section 4.2, this lazy evaluation is beneficial when boxes share enormous object graphs but only touch a fraction of the objects. Using lazy evaluation, Jigsaw never devotes computational resources to protect objects that are shared but never accessed.

For each public method belonging to `obj`, the surrogate defines a wrapper function whose `this` pointer is bound to `obj`. Thus, even if external code assigns the surrogate method to be a property on another object, the method will always treat `obj` as its `this` pointer. This prevents attacks in which a malicious box subverts a method's intended semantics by supplying an inappropriate `this` object.

When a surrogate function is invoked, it calls `createSurrogate()` on all of its arguments before passing those arguments to the underlying function. The surrogate function also calls `createSurrogate()` on the underlying function's return value, and returns that surrogate object to the original caller of the surrogate function.

In summary, surrogates automatically protect cross-box data exchanges. Since principal objects are surrogates, and boxes can be accessed only via their principal objects, Jigsaw ensures that all cross-box interactions respect public/private semantics.

2.6.3 Predefined JavaScript Objects

The browser predefines a set of JavaScript objects that live in each box's JavaScript namespace. The most important predefined object is the DOM tree; others provide support for regular expressions, mathematical functions, and so on. Jigsaw virtualizes the DOM tree in each box (§3), redirecting the box's DOM operations to a Jigsaw-controlled data structure that performs security checks before reflecting operations into the real DOM. Jigsaw also ensures that constructor functions for globally shared built-in objects like regular expressions are private and immutable. These safeguards prevent a malicious box from arbitrarily manipulating the visual display, or redefining constructor functions that are used by all boxes.

2.6.4 Cross-box Events

Box X may wish to register one of its functions as an event handler in a different box Y. To do so, X simply passes the handler to Y via Y's public interface; Y can then register the handler with the browser's event engine in the standard way. When the relevant event in Y occurs, the browser executes the handler like any other. However, the browser passes a scrubbed event object to the handler. This event does not contain references to Y's private-by-default JavaScript namespace. This prevents information leakage via foreign event handlers. Like all foreign methods, the handler executes in the JavaScript context of the box that created it.

2.7 Client-side Communication Privileges

A parent can restrict the principal objects that are visible to a child. By default, a child can reference its parent's principal object (`Jigsaw.getParentPrincipal()`), the principal objects of its own children (the `Jigsaw.principals` array), and the principal objects of other boxes from its own domain (`Jigsaw.getSameDomainPrincipals()`). Jigsaw associates each principal with a unique id, and a parent can restrict a child's access to a subset of principals ids.

2.8 Dropping Privileges

Jigsaw allows a box to voluntarily restrict the networking privileges that it received from its parent. A box can also relinquish the right to a visual field, or abandon the ability to write to DOM storage. Privilege can drop only in a monotonically decreasing fashion. For example, if a parent gives a child unrestricted network access, the child cannot restrict its privileges to only `foo.com` and then unrestrict itself.

2.9 Summary

Jigsaw provides a robust encapsulation technique for cross-principal sharing. All data is accessible by reference, but all data is implicitly hidden from external parties unless it is explicitly declared as `public` by the owning principal. Parents define resource permissions for the execution contexts of their children. Such permissions become monotonically stricter as the principal nesting depth increases.

Jigsaw does enforce some restrictions on the standard JavaScript language. In particular, a surrogate never exposes the prototype object or constructor function for the underlying object. Jigsaw also prevents box code from tampering with the prototypes of global built-in objects like Array. While this prevents boxes from changing externally defined prototype chains, well-written JavaScript code rarely uses such tricks, and allowing such behavior allows malicious boxes to launch prototype poisoning attacks [1, 11] against other boxes. Despite these restrictions, Jigsaw preserves many features of the standard JavaScript language. For example, Jigsaw supports closures, first-class function objects, object literals, an event-driven programming model, and pass-by-reference semantics for all objects, not just those shared within the same domain.

3 Implementation

Our Jigsaw implementation consists of a Jigsaw-to-JavaScript compiler and a client-side JavaScript library. The compiler parses Jigsaw code using an ANTLR toolchain [20]. A custom C# program adds static security checks to the resulting ASTs, and then translates the modified ASTs to JavaScript code that a browser can execute. The emitted JavaScript code also contains the client-side Jigsaw library, which implements runtime security checks and defines box management interfaces like `Jigsaw.createBox()`.

The rewriter modifies every object creation so that each object receives a unique integer id. This allows the Jigsaw library to maintain a mapping from raw objects to the associated surrogates, ensuring that `createSurrogate()` makes at most one surrogate for each raw object.[1]

The rewriter also tags each object with the id of its creating box (the Jigsaw library defines an internal `getCurrentBoxId()` function to which the rewriter can insert a call). This tag allows the Jigsaw library to determine whether a surrogate is being passed to its original box—if so, Jigsaw "unwraps" the surrogate, returning the backing object instead of the surrogate (§2.6.2). To allow `createSurrogate()` to determine the currently executing box context, the rewriter modifies all function definitions such that on entry, a function pushes its box id onto a stack, and on exit, a function pops the stack.

The rewriter translates `public` and `private` property declarations into operations on a per-object map of visibility metadata. The rewriter assigns such a map to each object at object creation time. Using property descriptors [19], Jigsaw ensures that per-object metadata is immutable and cannot be modified by a malicious or buggy box.

The Jigsaw library is responsible for creating new boxes. To do so, the library uses an `eval()` statement to dynamically load the (rewritten) box code. However, the `eval()` call is invoked within the context of a special Jigsaw function that defines aliasing local variables for standard global properties like `window`, `document`, and so on. The Jigsaw-defined aliases implement a virtualized browser environment that forces a box's communication with the outside world to go through Jigsaw's security validation layer. For example, Jigsaw's virtual `XMLHttpRequest` object ensures that a box's AJAX requests satisfy the security policies defined by the box's parent. Similarly, the virtual DOM tree is backed by a branch of the real DOM tree, but virtual operations are not reflected into the real tree unless they satisfy the parent box's security policy. For dangerous functions like `eval()`, and for sensitive internal Jigsaw functions, Jigsaw creates null virtualizations that do nothing or throw exceptions on access. As with all things JavaScript, there are various subtleties in the virtualization process that we elide due to space constraints.

Our current Jigsaw prototype implements the bulk of the design from Section 2. The primary exception is full DOM tree virtualization. This is still a work-in-progress due to the complexity of the DOM interface.

[1] To prevent this data structure from hindering garbage collection, Jigsaw requires weak maps, whose design is being finalized for the next version of JavaScript [16].

4 Evaluation

Jigsaw's goal is to provide an efficient, developer-friendly isolation framework. In this section, we describe our experiences with porting preexisting JavaScript libraries to Jigsaw; we then evaluate the performance of the modified libraries. We show that porting legacy code to Jigsaw is straightforward, that Jigsaw's pass-by-reference surrogates are much more efficient than pass-by-value marshaling, and that Jigsaw's dynamic security checks are similar in performance to those of other rewriting-based systems like Caja [15].

4.1 Porting Effort

We found that many preexisting JavaScript libraries already had an implicit notion of "public" and "private"; thus, porting these libraries to Jigsaw primarily consisted of making implicit visibility settings explicit via the `public` and `private` keywords. For example, at initialization time, many libraries add a single new object to the global JavaScript namespace, and use that object's properties as the high-level interface to the library code. This object serves as a de facto principal object (although it has none of the security protections afforded by Jigsaw). To port libraries like this to Jigsaw, we first declared the de facto gateway object to be the Jigsaw principal object for the library. We marked that object's properties with the `public` keyword. We then identified the public properties of other library objects with the help of an instrumented version of the Jigsaw runtime. For each surrogate that crossed a box boundary, the instrumented runtime logged the public and private properties for the surrogate's backing object; the runtime also modified each surrogate so that all foreign box accesses to private properties on the backing object threw an immediate exception instead of returning `undefined`.

The surrogate log and the fail-stop exceptions on private property accesses made it easy to identify legacy code properties that needed to be marked as public. Porting was also simplified because we did not have to worry about asynchronous control flows as in PostMash [2].

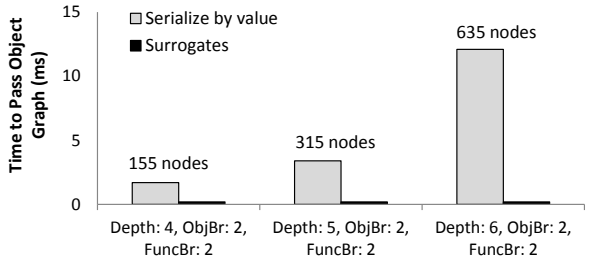

Figure 7: When mutually distrusting domains must share objects with each other, Jigsaw's pass-by-reference surrogate mechanism is much faster than pass-by-value marshaling (`Depth`: depth of shared object tree, `ObjBr`: Number of object properties per object, `FuncBr`: Number of function properties per object).

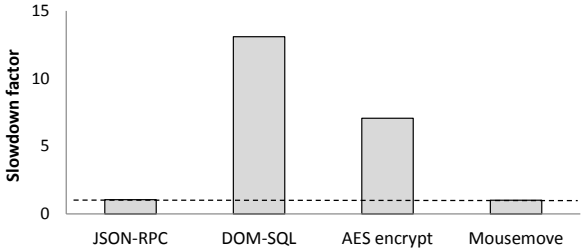

Figure 8: Jigsaw's extra security tests add between 0x–12x performance overhead (the dotted line indicates a slowdown factor of 1, i.e., no slowdown). Jigsaw's performance overheads are similar to those [10, 22] of other rewriting-based mashup frameworks like Caja [15].

We also did not need to explicitly insert object sanitization calls—in contrast to Caja [15], which requires developers to invoke a `tame()` sanitization function wherever an object crosses a trust boundary, Jigsaw automatically creates surrogates when "raw" objects travel between boxes. We provide a more detailed comparison of Jigsaw, PostMash, and Caja in Section 5.

4.2 Performance

In this section, we use microbenchmarks and Jigsaw-modified applications to evaluate Jigsaw's performance overheads. All of the results were generated on a Windows 7 PC with 4 GB of RAM and a dual-core 3.2 GHz processor. All web pages were executed in the Firefox 8.0.1 browser.

Sharing overheads: In mashup frameworks that use `postMessage()`, isolation containers share data by asynchronously exchanging immutable strings. If containers wish to share more complex objects, each endpoint must implement a marshaling protocol that serializes objects on sends and deserializes objects on receives. The marshaling costs can be high, particularly if containers wish to share functions, since the recipient has to dynamically compile each shared function's source code using `eval()`.

When a sender passes an object to a recipient, the sender actually shares an object graph that includes all of the objects that are recursively reachable from the root. Figure 7 depicts the sharing cost for synthetic object graphs of various sizes. In these experiments, the graphs were trees. The "Depth" metric corresponds to the height of the tree. The "ObjBr" (object branch) metric indicates how many of an object's properties referenced other objects; similarly, the "FuncBr" (function branch) metric indicates how many properties pointed to functions. Functions had no child objects or functions, i.e., they were leaf nodes in the object tree.

As Figure 7 demonstrates, passing an object graph between isolation containers using surrogates has almost zero overhead. When the root of an object graph is passed across a box boundary, Jigsaw must create a new surrogate object for it; however, the only cost is an object creation (for the surrogate object itself) and a function call for each public property on the backing object (to create the getter/setter property descriptors (§2.6.2)). In contrast, marshaling pass-by-value data is much more expensive, since the entire object graph must be traversed, serialized, and then deserialized, with costly `eval()` operations on the receiver-side to recreate shared functions. Note that the results in Figure 7 do not include `postMessage()` overhead, i.e., the sender and the receiver were in the same frame. Thus, these results are a conservative estimate of the marshaling costs in a `postMessage()` system.

In Jigsaw, passing large object trees across isolation barriers is efficient because surrogates are lazily created when a box actually tries to access a foreign object. Thus, at sharing time, initially only one surrogate must be created for the root of the object graph. In pass-by-value systems, once an object graph has been recreated by the receiver, property accesses on the graph are just as cheap as regular property access on locally created objects. In contrast, accessing a property through a Jigsaw surrogate introduces the overhead of invoking a getter or setter. Such accesses are roughly 30 times more expensive than a regular property access. However, we believe that this cost is acceptable for three reasons. First, it is only paid upon accessing external objects; it is not paid for objects that are never accessed by external domains, nor is it incurred when a box accesses its locally declared objects. Second, since Jigsaw's initial sharing cost is *O(1)* in the size of the object graph to share, Jigsaw reduces the initial sharing costs of non-trivial graphs by multiple orders of magnitude compared to a pass-by-value solution. Modern web applications already have object graphs containing

hundreds of thousands of objects [14], so for complex mashups exchanging complex object graphs, the initial sharing cost of pass-by-value will be unattractive due to the computational latencies. Third, unlike pass-by-value systems, surrogates allow mashups to communicate synchronously using pass-by-reference semantics. This makes developing mashups much easier, and makes Jigsaw's property access penalties easier to bear.

End-to-end performance: In addition to performing checks during property accesses, Jigsaw must perform a variety of bookkeeping tasks, e.g., assigning box ids and object ids to newly created objects, and interposing on accesses to virtualized browser resources to ensure that boxes adhere to the security policies established by their parents. Figure 8 shows the end-to-end performance slowdown for several Jigsaw-enabled applications. The slowdown is normalized with respect to the performance of the baseline, non-Jigsaw-enabled applications. The dotted line indicates a slowdown of 1, i.e., a situation in which Jigsaw adds no performance overhead. We examined the following applications:

- JSON-RPC [9] is a JavaScript library that layers an RPC protocol atop an AJAX connection. We defined the performance of a JSON-RPC session as the integrator-perceived completion rate of null RPCs that performed no actions at a localhost RPC server. Using a localhost server instead of a remote one eliminated the impact of network delay and allowed us to focus on Jigsaw's CPU overhead.
- The DOM-SQL [3] library provides a SQL interface to DOM storage. We defined the performance of DOM-SQL as the number of rows that the integrator could insert into a table per second. Each insertion caused a synchronous write to DOM storage.
- The AES encryption function belongs to the Stanford JavaScript crypto library [21]. Performance was defined as the throughput with which the integrator could feed plaintext to the library and receive ciphertext.
- Mousemove is a simple benchmark library which registers a handler for mouse movement in the integrator's DOM. A human user moves the mouse back and forth as quickly as possible, and the library increments a counter every time that its handler fires. Performance was defined as the number of times that the handler fired.

In the Jigsaw version of each application, the library code was placed in a separate box from the integrator, and all integrator-integratee communication took place through a principal object or a virtualized DOM resource.

Figure 8 shows that Jigsaw's security checks cause a 0-12x slowdown. The slowdown is application-dependent. For example, in the Mousemove test, the rate at which the browser fired mouse handlers was slow enough that Jigsaw's security overhead was hidden. The JSON-RPC test was similar, since the rate at which AJAX callbacks fired was also slow enough to hide Jigsaw's overhead. In contrast, in the encryption test and the DOM-SQL test, the applications were rarely blocked on external browser activity. Instead, these programs executed many small, application-defined functions. This incurred a lot of Jigsaw bookkeeping overhead, since Jigsaw had to update its internal call stack for each function invocation and return (§3). Nevertheless, Jigsaw's performance overheads were similar to those of other rewriting systems like Caja [10, 22].

5 Related Work

There are many preexisting frameworks for securing mashup applications. As we describe in more detail below, these systems present very different programming models to developers. At a high level, Jigsaw differs from all of these systems due to its focus on providing simple, efficient isolation mechanisms. Jigsaw is simple because it defines a concise ACL language for browser resources, a straightforward `public`/`private` distinction for JavaScript properties, and an automatic surrogate mechanism that transparently protects cross-domain data exchanges while preserving synchronous function call semantics. Jigsaw is efficient because its pass-by-reference surrogates avoid the marshaling overhead that afflicts pass-by-value systems.

ADsafe, FBJS, and Dojo Secure: ADsafe [6] and Dojo Secure [25] use a language subsetting approach, forcing guest code to be written in a restricted portion of the larger JavaScript language. In contrast, Jigsaw allows guest code to be written in a larger, more expressive subset. This makes it easier for developers to port legacy applications to Jigsaw (and write new Jigsaw applications from scratch). However, Jigsaw does pay a performance penalty due to the dynamic security checks that are required to secure the larger language subset.

FBJS [8] uses rewriting to prepend guest code identifiers with a unique random prefix. This effectively isolates the guest code from the integrator. FBJS allows guest code to interact with its parent through a restricted, virtualized API, e.g., through calls to a `VirtDOMnode.getParentNode()` method instead of through direct accesses to the parent's `DOMnode.parentNode` property.

Broadly speaking, FBJS, ADsafe, and Dojo Secure present a similar architectural model: strict guest isolation with a narrow, predefined interface between isolation containers. In contrast, Jigsaw allows principals to define their own public interfaces.

Caja: Like Jigsaw, Caja [15] is a rewriting system that places scripts inside virtualized execution environments. Caja defines a `tame(obj)` function that makes `obj` safe to pass to untrusted JavaScript contexts. `tame()` performs many of the security checks described in Section 2.6.2. For example, it prevents dynamic prototype manipulation, and it ensures that methods cannot be called with arbitrary `this` references.

Jigsaw differs from Caja in several important ways. In Jigsaw, objects and their properties are invisible to external domains by default. Developers use the `public` keyword to mark an interface as externally visible; visibility annotations are defined as part of the interface declaration. In contrast, Caja's visibility metadata is managed at interface *sharing* time instead of interface *declaration* time. In Caja, programmers must remember to invoke `tame(obj)` before `obj` is passed across an isolation barrier. This makes a program's security properties more difficult to understand, since developers can no longer reason about how an object can be accessed without inspecting all of the places at which the object crosses an isolation boundary. In contrast, Jigsaw's declaration-time visibility modifiers provide clearer, more centralized indications of object access rights. Jigsaw's surrogate mechanism also provides automatic "taming" as objects flow between boxes. This eliminates the developer burden of having to manually tame objects at sharing time. It also guarantees that taming takes place all of the time, regardless of whether the developer remembered to tame an object.

Also note that Caja's primary goal is to make it easy for an integrating page to incorporate untrusted scripts—guest-to-guest interactions are of secondary importance. Thus, host-to-guest communication is straightforward, but guest-to-guest interactions must be mediated by the host. This requires the integrator to define and manage a shared communication infrastructure. In contrast, Jigsaw's goal is to make it easy for arbitrary execution contexts to communicate through restricted interfaces. Using principal objects, any two contexts in Jigsaw can exchange information. Furthermore, the integrator is no longer responsible for managing cross-script communication. Instead, each script implements its own cross-box protocols.

Secure ECMAScript (SES): Secure ECMAScript (SES) [17] uses newly standardized features of ECMAScript 5 [7] to provide Caja-like isolation without requiring Caja-like rewriting and runtime virtualization. By pushing dynamic security checks into the JavaScript engine, SES can potentially offer dramatic reductions in the costs of these checks. SES is still being formulated, but once ECMAScript 5 becomes widespread, Jigsaw can use SES techniques to implement its security abstractions.

Pass-by-value systems: Systems like PostMash [2] use iframes as isolation containers, and implement cross-domain communication using the asynchronous, pass-by-value `postMessage()` call. Such isolation frameworks have several drawbacks. First, there is significant marshaling overhead if domains share non-trivial object graphs (§4.2). To avoid this penalty, domains can keep local copies of object graphs and synchronize views across iframes. However, this approach still requires frequent exchanges of synchronization messages. In PostMash, this message traffic induced a 60% performance decrease in a Google Maps mashup [2].

A second drawback of these systems is that they rely on an asynchronous channel for cross-domain communication. Asynchronous communication is an awkward fit for many mashup applications; for example, it is ill-suited for the integration of computationally intensive modules that implement databases, cryptographic operations, input sanitizers, image manipulation routines, and so on. Rewriters can transform asynchronous operations into quasi-synchronous ones using continuation-passing [11]. However, many programmers find it difficult to reason about continuations. Furthermore, continuations can introduce subtle race conditions that are not present in truly synchronous environments (§2.1).

Browsers give each iframe a separate execution thread. Thus, iframe isolation does have the advantage that a parent can make forward progress if a child is hung or computationally intensive. In Jigsaw, each box resides within the same iframe, so a misbehaving child can intentionally or inadvertently perform a denial-of-service attack on its parent. We do not view this as a major disadvantage of Jigsaw, since modern browsers allow users to terminate unresponsive scripts via a pop-up warning dialog.

Object views: Object views [11] let developers specify policy code that controls how objects are shared across isolation boundaries. Policy code is written in the full JavaScript language and is attached to view objects that mediate external access to private backing objects. This security model is very expressive, but we believe that it is unnecessarily complex (and therefore error-prone). Jigsaw's `public` and `private` modifiers present a more intuitive programming model, allowing developers to express simple "yes-no" disclosure policies. In contrast, when a developer writes an object view policy, she must reason about the execution context that initiates an access request, and how context-specific factors should influence data exposure. Jigsaw's visibility identifiers act as explicit, declaration-time indications of visibility policy.

IFC: Information flow control (IFC) systems like Jif [13] assign security labels to variables, allowing developers to precisely specify the data that principals

should be allowed to read or write. We eschewed an IFC mashup architecture for two reasons. First, a well-known problem with IFC is that programmers are loath to generate the required security annotations. In contrast, simple visibility modifiers like `public` and `private` have not triggered a similar level of complaint. IFC-style labels are also ill-suited for governing access to browser resources. For example, it is difficult to use labels to express policies like "give a principal update rights to the leftmost 30% of the visual display." Jigsaw can easily express such a policy using a simple CSS-style rule.

ConScript: ConScript [12] uses a modified browser engine to enforce security. Integrators restrict the behavior of guests by attaching policy code to key execution points, e.g., the invocation of a function or the loading of a new script. Like object view policies, ConScript policies are written in arbitrary JavaScript and can be extremely expressive. However, as mentioned above, Jigsaw strives to provide simple, developer-friendly security policies, and we have found that in practice, Jigsaw's simpler policies are sufficient to express many kinds of mashup architectures.

OMash: Like Jigsaw, OMash [5] allows each principal to define a public set of functions that other principals can invoke. However, OMash does not have a private-by-default visibility policy, nor does it wrap objects in proxies before handing them to external domains. Thus, public OMash functions expose an ostensibly narrow interface, but their return values can expose sensitive data. For example, the caller of a public OMash function can perform arbitrary JavaScript reflection on the properties of the returned object (and any other objects reachable from that root). If the caller modifies any of this data, the modifications will be visible in the data's source domain.

MashupOS: MashupOS [23] provides a new set of isolation abstractions for web browsers. In MashupOS, a service instance is a browser-side analogue of a traditional OS process. Each instance gets a partitioned set of hardware resources like CPU and memory, and communicates with other instances using asynchronous, pass-by-value messages. Jigsaw eschews such a communication style in favor of synchronous, pass-by-reference messaging. This necessitates a mechanism like surrogates (§2.6.2) for securely exchanging objects across isolation boundaries.

CommonJS Modules: CommonJS [4] defines a module system for JavaScript. CommonJS gives each library a protected namespace and the ability to define external interfaces. However, CommonJS namespaces are implemented using closures. Thus, unlike Jigsaw boxes, CommonJS namespaces do not protect against attacks like prototype poisoning [1, 11]. CommonJS also does not provide strong notions of public and private data. Thus, as in OMash, the return values from public functions can inadvertently leak private module data.

6 Conclusion

Jigsaw is a new mashup framework for web applications. It allows mutually distrusting content providers to define narrow public interfaces for their private client-side state. Jigsaw strives to be developer-friendly, so it eschews the complicated security policies of prior mashup frameworks; instead, Jigsaw uses the `public` and `private` keywords to mark data as externally visible or domain-private. Jigsaw's security semantics are thus easily understandable to programmers who are familiar with popular languages like Java that also use `public`/`private` distinctions.

Prior mashup frameworks often isolate state using iframes or iframe-like abstractions. These isolation containers force domains to communicate using asynchronous pass-by-value channels. In contrast, Jigsaw's novel surrogate mechanism allows domains to pass objects by reference using synchronous function calls. This makes it easier for developers to reason about cross-origin sharing, since accessing a locally defined object or function looks no different than accessing an object or function that has been shared by an external domain. Pass-by-reference surrogates are also more efficient than pass-by-value approaches because surrogates do not incur marshaling overhead when they travel between domains.

Our evaluation shows that existing web applications are easily ported to the Jigsaw framework. Our evaluation also demonstrates that Jigsaw has similar or better performance than prior mashup schemes.

Acknowledgments

We would like to thank David Wagner and the anonymous reviewers for their comments on earlier versions of this paper. This material is based upon work partially supported by a NSF Graduate Research Fellowship. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] ADIDA, B., BARTH, A., AND JACKSON, C. Rootkits for JavaScript Environments. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT)* (2009), USENIX Association.

[2] BARTH, A., JACKSON, C., AND LI, W. Attacks on JavaScript Mashup Communication. In *Web 2.0 Security and Privacy* (2009).

[3] BOERE, P. dom-storage-query-language: A SQL inspired interface for DOM Storage. http://code.google.com/p/dom-storage-query-language/.

[4] COMMONJS. Modules/1.1.1 Specification, February 2012. http://wiki.commonjs.org/wiki/Modules/1.1.1.

[5] CRITES, S., HSU, F., AND CHEN, H. OMash: Enabling Secure Web Mashups via Object Abstractions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (2008), ACM, pp. 99–108.

[6] CROCKFORD, D. ADsafe. http://www.adsafe.org.

[7] ECMA INTERNATIONAL. Standard ECMA-262: ECMAScript Language Specification, 5.1 Edition, June 2011. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf.

[8] FBJS (FACEBOOK JAVASCRIPT). http://developers.facebook.com/docs/fbjs.

[9] GHERARDI, G. jsonrpcjs. https://github.com/gimmi/jsonrpcjs.

[10] GOOGLE. google-caja: Performance of cajoled code. http://code.google.com/p/google-caja/wiki/Performance.

[11] MEYEROVICH, L., FELT, A., AND MILLER, M. Object Views: Fine-grained Sharing in Browsers. In *Proceedings of the 19th International Conference on World Wide Web* (2010), ACM, pp. 721–730.

[12] MEYEROVICH, L., AND LIVSHITS, B. ConScript: Specifying and Enforcing Fine-grained Security Policies for JavaScript in the Browser. In *Proceedings of the IEEE Symposium on Security and Privacy* (2010), IEEE, pp. 481–496.

[13] MEYERS, A. C., ZHENG, L., ZDANCEWIC, S., CHONG, S., AND NYSTROM, N. Jif: Java Information Flow, July 2001. http://www.cs.cornell.edu/jif.

[14] MICKENS, J., ELSON, J., HOWELL, J., AND LORCH, J. Crom: Faster Web Browsing Using Speculative Execution. In *Proceedings of NSDI* (2010).

[15] MILLER, M., SAMUEL, M., LAURIE, B., AWAD, I., AND STAY, M. Caja: Safe active content in sanitized JavaScript. Google white paper. http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf.

[16] MILLER, M. S. ES Wiki: harmony:weak_maps, May 2011. http://wiki.ecmascript.org/doku.php?id=harmony:weak_maps.

[17] MILLER, M. S. SES (Secure EcmaScript), May 2011. https://code.google.com/p/es-lab/wiki/SecureEcmaScript.

[18] MIX, N. Narrative JavaScript. http://www.neilmix.com/narrativejs/doc/.

[19] MOZILLA DEVELOPER NETWORK. Object.defineProperty(). https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/defineProperty.

[20] PARR, T. *The Definitive ANTLR Reference*. Pragmatic Bookshelf, Raleigh, North Carolina, 2007.

[21] STARK, E., HAMBURG, M., AND BONEH, D. Symmetric Cryptography in JavaScript. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2009), IEEE, pp. 373–381.

[22] SYNODINOS, D. ECMAScript 5, Caja and Retrofitting Security: An Interview with Mark S. Miller, February 25 2011. http://www.infoq.com/interviews/ecmascript-5-caja-retrofitting-security.

[23] WANG, H., FAN, X., HOWELL, J., AND JACKSON, C. Protection and Communication Abstractions for Web Browsers in MashupOS. In *Proceedings of SOSP* (October 2007).

[24] WANG, H., GRIER, C., MOSHCHUK, A., KING, S., CHOUDHURY, P., AND VENTER, H. The Multi-Principal OS Construction of the Gazelle Web Browser. In *Proceedings of the 18th USENIX Security Symposium* (2009), USENIX Association, pp. 417–432.

[25] ZYP, K. Secure Mashups with dojox.secure, August 2008. http://www.sitepen.com/blog/2008/08/01/secure-mashups-with-dojoxsecure/.