Protection and Communication Abstractions for Web Browsers in MashupOS

Helen J. Wang  Xiaofeng Fan  Collin Jackson  Jon Howell
helenw@microsoft.com  xiaoffan@microsoft.com  collinj@cs.stanford.edu  howell@microsoft.com

Abstract—Web browsers have evolved from a single-principal platform on which users browse one site at a time into a multi-principal platform on which data and code from mutually distrusting sites interact programmatically in a single page on the client side, enabling feature-rich “Web 2.0” applications (or mashups) that offer close-to-desktop experiences. However, the protection and communication abstractions offered by today’s browsers remain suitable only for a single principal system—either no trust through complete isolation between principals (sites) or full trust by incorporating third party code as libraries.

In this paper, we address this deficiency by providing the missing abstractions to enable robust and secure Web applications on a rich multi-principal platform. We draw an analogy between Web sites’ sharing of browser resources and users’ sharing of operating system resources, and we use this analogy as a guide to develop protection and communication abstractions for Web browsers. Our abstractions are designed to match the trust relationship in Web service provisioning and integration so that they can enhance both security and ease in Web service creation. For example, our abstractions can be used to prevent Cross Site Scripting (XSS) vulnerabilities in Web services in a fundamental way, addressing a prominent threat in today’s Web. Furthermore, we have designed our abstractions to be backward compatible and easily adoptable. We have built a prototype system that realizes almost all of our abstractions and their associated properties. Our evaluation shows that our abstractions make it easy to build more secure and robust client-side Web mashups and can be easily implemented with negligible performance overhead.

I. INTRODUCTION

Web browsers are becoming the single stop for everyone’s computing needs including information access, personal communications, office tasks, and e-commerce. Today’s Web applications synthesize the world of data and code, offering rich services through Web browsers and rivaling those of desktop PCs. Browsers have evolved to be a multi-principal operating environment where mutually distrusting Web sites (as principals) interact programmatically in a single page on the client side, sharing the underlying browser resources. This also resembles the PC operating environment where mutually distrusting users share host resources.

However, unlike PCs that utilize multi-user operating systems for resource sharing, protection, and management, today’s browsers do not employ any operating system abstractions, but provide just a limited binary trust model and protection abstractions suitable only for a single principal system: There is either no trust across principals through complete isolation or full trust through incorporating third party code as libraries. Consequently, Web programmers are forced to make tradeoffs between security and functionality, and often times sacrifice security for functionality.

A. Goals

In the MashupOS1 project, we aim to design and build a browser-based multi-principal operating system. Among the myriad of operating system issues, we focus on the most imminent needs of today’s browsers: abstractions for protection and communication. The goal of protection is to prevent one principal from compromising the confidentiality and integrity of other principals, while communication allows them to interact in a controlled manner.

We follow the principles below in designing our abstractions for Web programmers:

- Match all common trust levels: We must understand all the common trust levels between Web service providers and integrators and aim to provide abstractions matching these levels of trust. Otherwise, Web programmers would face making tradeoffs among trust levels, either trusting more, and sacrificing security, or trusting less, and losing functionality.

- Strike a balance between ease-of-use and security: One might argue that a system is either secure or insecure and there should be no middle ground. This is true for designing a security system like an authentication system, but not true when designing abstractions for programmers. A rigid set of abstractions that tie programmers hands and limit flexibility for the purpose of better security often has short life span, since programmers can build up libraries on the rigid interfaces to make their lives easy and make them a de-facto abstraction for all programmers. It would be better for abstraction designers to design those abstractions with security in mind. Our goal here is to provide a full set of abstractions and enable programmers to build robust and secure services that match their trust expectations (see above bullet), rather than to prevent programmers from shooting themselves in the foot.

- Easy adoption and no unintended behaviors: Allowing easy adoption is paramount in our abstraction design. We must ensure that our abstractions allow programmers to provide fallback mechanisms when Web pages using them are rendered by legacy browsers. We also must

1 We use this name to fulfill the anonymity requirement of SOSP submission.
ensure that there are no undesirable interactions between new services and old services in the new browser environment where our abstractions are supported.

B. Contributions

This paper characterizes the trust relationships between Web service providers and integrators, and identifies trust relationships that are not well served by contemporary abstractions. We present two new isolation abstractions to address these shortcomings. We present a mashup communication model that unifies recent proposals. We also introduce the notion of restricted services, which enable Web sites to host rich third-party content without exposing themselves to Cross Site Scripting attacks.

We have designed these new abstractions to be backward compatible and be free of undesirable interactions with legacy browsers and legacy services. Our prototype implementation and evaluation demonstrate that the abstractions can be practically integrated into modern browser software with negligible overhead.

C. Outline

For the rest of the paper, we first give background in Section II presenting the evolution of Web browsers and the inner workings of current browsers. In Section III, we describe browser resources and define MashupOS principals that own these resources. We describe the possible trust relationships between MashupOS principals in Section IV. In Section V, we present our proposed abstraction of the units of isolation, fault containment, and resource sharing, and describe our proposal for communication. In Section VI, we provide a solution for service providers to host untrusted third-party content, and we present a sandboxing abstraction as a first class abstraction for Web programmers. In Section VII, we show how our isolation and sandboxing abstractions can be utilized to combat Cross Site Scripting attacks, which pose a major threat to dynamic, interactive Web applications. We have built a MashupOS prototype based on Internet Explorer and realized almost all our proposed abstractions and their properties, demonstrating that our abstractions can be easily added to existing browsers. We describe our implementation in Section VIII. In Section IX, we present a showcase Web application combining Google Maps and a Flickr geo-tagged photograph gallery to show the location of the photograph taken and evaluate the performance overhead of our prototype system. In Section X, we compare and contrast with related work, and finally we conclude in Section XI.

II. BACKGROUND

The Web has evolved from a collection of static documents connected by hyperlinks into a dynamic, rich, interactive experience driven by client-side code and aggregation by Web services. The security policy of modern browsers was designed primarily to avoid vulnerabilities in old sites, rather than providing the best abstractions for the newest sites. In this section, we summarize the existing access control policies and the limitations they place on Web site design, then describe a new access control policy that mashup authors are demanding through existing proposals.

A. Same-Origin Policy

As Web pages became more dynamic in the 1990s, browsers introduced cookies as a way to differentiate users and to operate in a way that depends on the user. Cookies add state to the otherwise stateless HTTP protocol, giving the user the illusion of isolated, continuous connections to remote sites, as shown in Figure 1. Cookies are small, arbitrary pieces of data (often simply a session identifier) chosen by the Web server and sent to the browser when responding to a request. They are returned to the server that sent them (and only that server) by the browser on subsequent requests [22].

Web applications built around cookies rely on the browser to keep their sessions isolated from other sites; if an attacker can make requests with a logged-in victim’s cookie, the attacker becomes indistinguishable from the victim. This security policy, enforced by the browser, is known as the Same-Origin Policy [18]:

**SAME-ORIGIN POLICY (SOP)**

Only the site that sends some information to the browser may later read or modify that information.

JavaScript, a more recent browser feature, allows code from one page or frame (subpage) to read and modify other pages or frames, using the Document Object Model (DOM), an interface that allows script to read and modify the content, structure, and style of HTML documents. These types of accesses across domain boundaries could allow the attacker to misuse the user’s cookie, or even read pages behind corporate firewalls that would not be directly accessible by the JavaScript’s author. To prevent these attacks and maintain isolation, the same-origin policy was applied to JavaScript, using a definition of “site” based on the scheme (http or https), DNS name, and TCP port of the URL [28]. For example, frames from http://a.com/ and http://b.com/ are isolated from each other — they cannot access each other’s resources including DOM objects and cookies.
The XMLHttpRequest method of data communication, which allows JavaScript to download XML files from its origin server and has become the ubiquitous tool of so-called “AJAX” (asynchronous JavaScript and XML) applications, is constrained by the SOP as well. For example, a frame from http://a.com/ cannot issue an XMLHttpRequest to http://b.com/.

B. Binary Trust Model

Web developers often wish to incorporate code from third-party sources, such as the Google Maps library. JavaScript files rarely contain sensitive information behind firewalls and are usually not access-controlled by cookies, so browsers implicitly assume that files in this format are public code libraries and allow them to be executed across domains, bypassing the SOP. The code runs as a library, with the privileges of the page including it. For example, a.com/service.html may contain the markup <script src='http://b.com/lib.js'/>, which allows lib.js to access a.com’s HTML DOM objects, cookies and data through XMLHttpRequest. However, lib.js cannot access b.com’s resources since lib.js is associated with the site a.com, but not b.com in this context.

This powerful but dangerous workaround to the ordinary isolation policy of the SOP manifests a binary trust model: a site either completely distrusts another site and is segregated through the use of cross-domain frames, or a site can use another site’s code as its own, offering full resource access to the remote site.

C. Web Mashups

Web mashups are defined as Web sites that compose data from more than one site, yet this definition is in tension with the same-origin policy, which prevents such communication. Many data providers want to publish information for any integrator site to use, but the same-origin policy prevents the browser’s XMLHttpRequest from loading the data directly.

 Initially, mashup developers worked around these restrictions using a proxy approach: taking information from a third party and making it appear to the browser to be “same-origin.” The pipes.yahoo.com mashup creation wizard is a recent example of this approach. When a Web user visits the mashup on pipe.yahoo.com, it connects to pipes.yahoo.com to get data. The request is proxied to the real data provider, like NYTimes, and the response data is then passed back from pipe.yahoo.com to the mashup. The drawback of this approach is that the content makes several unnecessary round trips, reducing performance, and the proxy can become a choke point, limiting scalability.

By encoding public data in executable JavaScript format (JavaScript Object Notation, or JSON [21]), cross-domain script tags can also be used to pass data from the provider to the integrator across domain boundaries, eliminating the need for proxies. This technique has the unfortunate side effect of granting the integrator’s privileges to the data provider. The binary trust model of the SOP forces the integrator to make tradeoffs between security and functionality.

D. Gadget Aggregators

Web gadget aggregators, such as Google Personalized Homepage [17] and Windows Live [23], are an advanced form of mashup, combining user-selected active content from third-party sources into a single portal page. A gadget is an HTML-plus-JavaScript component designed to be included into a gadget aggregator page; it is the client side of some Web service.

Gadget aggregators are security-conscious; they host each untrusted gadget in a frame on a distinct (sub)domain, relying on the SOP to isolate third-party gadgets from one another and from the outer page. However, because the SOP prevents interoperation among gadgets, aggregators also support inline gadgets, which include third-party code as a library of the aggregator page using the script tag. The binary trust model of today’s browsers unfortunately forces the gadget aggregator to decide between interoperation and isolation. Because inlining requires complete trust, Google’s aggregator punts the security problem to the user: “Inline modules can... give its author access to information including your Google cookies... Click OK if you trust this module’s author.”

E. Next-Generation Communication Proposals

Frustrated by the limitations of the SOP, mashup developers have been pushing browser vendors to move to a new security policy for the Web, allowing fine-grained policy decision making between communicating domains. Web service providers want to be able to make their own access control decisions as to whether data they send between domains is public, whether it should be executed, or whether it should be ignored, rather than relying on the browser to make this decision. Several new browser communication proposals [6], [12] have emerged, governed not by the SOP, but rather by a new type of policy that we call the verifiable-origin policy:

**Verifiable Origin Policy (VOP)**

A site may request information from any other site, and the responder can check the origin of the request to decide how to respond.

Communication that obeys the VOP is an important building block of our MashupOS proposal, and is discussed further in Section V-D.

III. PRINCIPALS AND RESOURCES

We cast the world of Web applications in the context of the conventional notion of the multi-user operating system, in which different *principals* have access to different sets of *resources*. In the OS environment, the principal is a user or group. By associating a process with a principal, the OS ensures that the process only has as much power as the principal that controls the process’ behavior. In general, one
user does not trust another with respect to the confidentiality and integrity of her resources.

In the Web environment, the principal is the owner of some Web content. In practice, browser adherence to the SOP means that the real notion of Web principal is tied to ownership of a DNS domain.

Other notions of principal have been explored. Before the SOP, the original cookie specification [22] allowed a page to restrict a cookie to only be sent to its server on subsequent requests if those requests were for pages starting with a particular path prefix. This access control policy does not match the trust hierarchy of most Web server deployments, since the owner of the root of a Web server ultimately controls all the subpaths, while the subpaths may contain less trustworthy, third-party content. With the advent of the SOP, the use of path-restricted cookies became a moot way to protect one page from another on the same server, since same-domain pages can directly access the other pages and pry their cookies loose.

An alternative notion of principal we considered allows applications to identify themselves using more of the URL than the SOP domain tuple; unlike the original cookie path hierarchy, this approach reflects the fact that the root of a Web server dominates its subpaths. The motivation for this alternative is the common Web practice of delegating ownership of Web server content by pathname, such as the ubiquitous ’/user/’ user home directory scheme used on departmental Web servers. Unfortunately, this scheme fails for the same practical reason as cookie paths: providing legacy support for the SOP’s coarser notion of principal destroys the distinction between path-based fine-grained principals. The practical consequence of the ubiquity of SOP-domain-based principals is that Web servers that wish to provide fine-grained separation of principals must do so using DNS subdomains rather than subpaths.

Because of the difficulty with escaping the SOP legacy, we preserve the idea of using the SOP domain (scheme, DNS host, TCP port) tuple as the principal in MashupOS. Therefore, the rest of the paper uses the terms “domain” and “principal” interchangeably.

Browsers provide to applications the following resources:

- **Memory**: The heap of script objects including HTML DOM objects that control the display. This is analogous to process heap memory.
- **Persistent state**: Browsers provide applications with the ability to store a few kilobytes of cookies or some other persistent state [16], which persist across application invocations. This resource is weakly analogous to the OS file system.
- **Network communications**: The ability to send and receive messages outside the application, equivalent to an OS network facility.

Later, we address how MashupOS assigns these resources to different domains, and to what extent domains are allowed to access resources belonging to other domains.

### IV. Trust Model among Principals

An important goal of our work is to design abstractions that match common trust levels of Web programmers in their service creation, whether as a service provider providing a Web service or component or as a service integrator integrating or composing others’ services into a mashup. Today’s browsers offer abstractions for only a binary trust model, which is insufficient for today’s Web services (Section II). In this section, we derive the common trust levels of Web programmers and point out the trust levels for which new abstractions are needed.

#### A. Services offered by Providers

Service providers want to segregate the data and code that they serve into three categories:

1) **Private, sensitive content** must be access-controlled by the service provider through a well-defined service API. For example, for a Web mail provider, a user’s mailbox and contact list are sensitive information; access to that information must be authenticated and authorized through a service API given by the Web mail provider. We call providers’ services that control access to their private, sensitive data and code access-controlled services.

2) **Public content** can be freely used by anyone. For example, a map provider may give away a code library for anyone to use for accessing its public map data. We call such services library services.

3) **Restricted content** is third-party content hosted by a provider, which the provider does not trust to access other private content from the provider’s domain. In today’s world, there is no distinction between restricted content and public content from the same domain. For example, a user profile hosted by a social networking Web site in the same domain is treated as public content. There is no way for the provider to indicate the untrustworthiness of such content and that browsers should deny such content’s access to any domain’s resources unless explicitly allowed. This deficiency has led to vulnerable Web services that suffer from devastating attacks like Cross Site Scripting (XSS) attacks (Section VII). We believe establishing this category of service and providing support for it is very much needed.

When restricted content is hosted privately and access-controlled by a provider, it becomes an access-controlled service (the first category).

For the security of its site, the provider must ensure that no matter how library services and restricted services may be used (or abused) by an integrator, it will not violate the access control of the provider’s access-controlled services. For example, a provider that offers both an access-controlled mail service and a public map library service must ensure that its map library code or any other third party restricted content have no access to any of its users’ mailbox and contact lists.

#### B. Trust Relationship between Providers and Integrators

Now, we analyze the trust relationship between an integrator and a provider at separate domains. Table I summarizes this
Today’s browsers only have abstractions for two trust levels: no trust through the use of a cross-domain frame and full trust through script inclusion (Section II). No trust is just one configuration of controlled trust. In the following sections, we will propose new abstractions ServiceInstance and Sandbox along with communication abstraction CommRequest to support controlled trust and asymmetric trust, respectively.

V. Isolation and Communication

A. Problems with existing browsers (and proposals)

Existing mainstream browsers have no mechanism for controlled cross-domain communication. The only cross-domain communication primitive available is the <script> tag, which gives the service provider uncontrolled access to the integrator’s domain. There do exist some cross-domain communication proposals (browser-to-server [1], [6], [12] and browser-side [31]), but they are not widely adopted, and they each exist in isolation, so none provides a general solution to the problem of mismatched trust patterns.

As a result of the lack of controlled cross-domain communication, there is also no way for a parent window and a child window, containing mutually untrusted content, to flexibly negotiate the layout of the boundary between them. Browser <frame>s offer isolation at the cost of rigid, parent-controlled layout; <div>s offer flexible, content-sensitive layout at the cost of requiring full trust between parent and child content.

Finally, the only protection abstraction in contemporary browsers is the SOP boundary. Unlike an OS process, a single principal cannot instantiate this abstraction multiple times to provide fault containment among multiple applications.

B. Isolation and Fault Containment: ServiceInstance

To solve these problems, MashupOS introduces the ServiceInstance abstraction, which enables an application or component from one domain to integrate a component from another domain, isolating the components while enabling controlled communication between them. This abstraction realizes the controlled trust scenario (Section IV).

An application instantiates a ServiceInstance with the tag:

```
<ServiceInstance
   src="http://alice.com/app.html" id="aliceApp">
```

The tag creates an isolated environment, analogous to an OS process, fetches into it the content from the specified src, and associates it with the domain alice.com that served that content.

To understand the value of the ServiceInstance abstraction, we must specify how resources are isolated. A ServiceInstance is a unit of resource allocation, to account for commodity resources such as CPU and memory pages, as well as a protection boundary, to prevent other domains from compromising the privacy or integrity of the data stored in those resources. This paper defers the issue of commodity resource allocation to future work, and focuses on the trust model.

### Table I

<table>
<thead>
<tr>
<th>Provider</th>
<th>Library service (Full access to public content)</th>
<th>Access controlled service (Controlled access to private content)</th>
<th>Restricted service (Full access to restricted content)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full access to me</td>
<td>1. Full trust</td>
<td>3. Controlled trust</td>
<td>5. Asymmetric trust</td>
</tr>
<tr>
<td>Controlled access</td>
<td>2. Asymmetric trust</td>
<td>4. Controlled trust</td>
<td>6. Asymmetric trust</td>
</tr>
<tr>
<td>to me</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Analysis.

The integrator may either trust the provider to fully access the integrator’s resources, or not trust the provider and export an access control service API for the provider to access the integrator’s resources. This corresponds to the two rows in Table I: “Full access” and “Controlled access”. (The table does not show how the integrator may control its own access to the provider because in this scenario, the integrator would serve as its own provider proxying the provider’s services.)

When the provider offers a library service, the integrator may access “full access”, then the integrator uses the library as its own code accessing the integrator’s resources, such as its HTML DOM objects, cookies, and obtaining remote data from the integrator’s Web server through XMLHttpRequest. This manifests a full trust between the integrator and the provider’s library service, as shown in Cell 1 in the table. If the integrator offers “controlled access”, then this manifests an asymmetric trust (Cell 2) where the integrator can access the library freely, but the library must use the integrator’s access control service API to access the integrator’s resources.

When the provider offers an access-controlled service, if the integrator offers “full access”, then the provider’s access control API dictates the resource access on both the provider and the integrator. This manifests a controlled trust where the provider trusts the integrator to the extent allowed by the provider’s access control policy for the integrator. This corresponds to Cell 3. If the integrator offers “controlled access”, the exchange of information between the integrator and the provider goes through two access control service APIs. This manifests bidirectional controlled trust, as shown in Cell 4. If we provide an abstraction for the scenario of a single direction controlled trust, then the bidirectional scenario simply requires two uses of the abstraction, one for each direction.

When the provider offers a restricted service, browsers should force the integrator to have at least asymmetric trust with the service regardless of how trusting the consumers are, as shown in Cell 5 and 6.
on the concerns of protecting memory, persistent state, and display resources.

1) Memory: Each ServiceInstance has its own isolated region of memory: No ServiceInstance can follow a JavaScript object reference to an object inside another ServiceInstance. This is true even for ServiceInstances associated with the same domain, just as multiple OS processes can belong to the same user: one domain can use ServiceInstances to provide fault containment among multiple application instances.

2) Persistent state: Cookies are handled no differently than in existing browsers: two ServiceInstances can access the same cookie data if and only if they belong to the same domain, just as two processes can access the same files if they are running as the same user.

3) Display: A raw ServiceInstance comes with no display resource. Instead, the parent ServiceInstance must allocate a subregion of its own display, called a Friv (more in the next section), and assign the Friv to the child ServiceInstance. The parent may use Friv to assign multiple regions of its display to the same child ServiceInstance, just as a single process can control multiple windows in a desktop GUI framework, such as a document window, a palette, and a menu pop-up window. As Web applications grow in sophistication, and as sophisticated Web “window managers” appear to manage these applications [32], it will be important for MashupOS to support this pattern.

C. Flexible Cross-Domain Display: Friv

The Friv is a flexible cross-domain display abstraction. Friv behaves like a conventional iframe in that it enables content to use part of its container’s display while otherwise isolating their resources. The iframe is difficult to use in tightly-integrated applications because the parent specifies the iframe’s size regardless of the contents of the iframe. Web developers prefer the div tag: because its contents and its container share a domain, the browser layout engine can resize the div’s display region to accomodate its contents.

The following tag syntax creates a new Friv and assigns it to an existing ServiceInstance.

<Friv width=400 height=150 instance="aliceApp">
This alternate syntax creates a new ServiceInstance and a new Friv simultaneously, and assigns the latter to the former.

<Friv width=400 height=150 src="http://alice.com/page.html">

The MashupOS Friv is so named because it crosses the iframe and the div. It isolates the content within, but it includes default handlers that negotiate layout size across the isolation boundary using the MashupOS local communication primitives (Section V-D). These handlers give the Friv the convenient div-like layout behavior.

1) ServiceInstance and Friv Life Cycle: The life cycle of a ServiceInstance is limited, by default, by the ServiceInstance’s responsibility for some part of the browser’s display. A ServiceInstance can track the display regions that it owns by registering a pair of handlers with the methods:
ServiceInstance.attachEvent(func, 'onFrivAttached');
ServiceInstance.attachEvent(func, 'onFrivDetached');

The first callback is invoked whenever the parent assigns a new Friv display to the ServiceInstance. When the parent reclaims the display associated with a Friv (by removing the Friv element from its DOM tree), the Friv’s display disappears from the child ServiceInstance’s object space, and the child’s onFrivDetached handler is called.

The default onFrivAttached and onFrivDetached handlers track the set of Frivs. When the last Friv disappears, the ServiceInstance no longer has a presence on the display, so the default handler invokes ServiceInstance.exit() to destroy the ServiceInstance.

A ServiceInstance can act as a daemon by overriding the default handlers so that it continues to run even when it has no Frivs. Such a ServiceInstance may continue to communicate with remote servers and local client-side components, and has access to its persistent state.

When a Friv is assigned to a new location (for example, using document.location = url, or equivalently, when the user clicks on a simple link in the Friv’s DOM), the Friv’s fate depends on the domain of the new location. The next two paragraphs describe the two possibilities.

If the domain is different from that of the ServiceInstance that presently owns the Friv, the behavior is just as if the parent had deleted the Friv (detaching it from the existing ServiceInstance) and created a new Friv and ServiceInstance with the <Friv src=...> tag. The only resource carried from the old domain to the new is the allocation of display real-estate assigned to the Friv. This behavior is analogous to creating a new process with a new identity, giving it the handle of the existing X Window region, and disconnecting the prior process from the same X Window.

If the domain matches that of the ServiceInstance that owns the Friv, then the HTML content at the new location simply replaces the Friv’s layout DOM tree, which remains attached to the existing ServiceInstance. Any scripts associated with the new content are executed in the context of the existing ServiceInstance.

Browsers allow a Web application to create a new “popup” window. The creation of a popup creates a new parentless Friv associated with the ServiceInstance that created the popup.

Given this definition of the life cycle of a ServiceInstance and Friv, the legacy <Frame> tag is implemented as follows: For each domain, there is a special “legacy” ServiceInstance. The <Frame src=x> tag is an alias for <Friv src=x instance=legacy>. Thus, all frame content and scripts for a single domain appear
in a common object space, just as they do in legacy SOP-only browsers. Within the legacy ServiceInstance, each script still has a local document reference that identifies the Friv whose DOM the script was loaded with, so that references like document.location are meaningful.

D. Communication: CommRequest

As shown in Figure 1, a Web application is the pairing of browser-side components and server-side components, all owned by a common domain. This model guides the MashupOS design of communication among domains: to the extent that a browser-side application is an extension of a server-side application, it should be allowed to communicate with other domains in the same ways that the server can (see Figure 2).

Legacy browsers follow the SOP in that they enable only communication from the browser-side component to its corresponding server (arrow 1). By communication we mean transfer of arbitrary data, such as an XML file. Servers, however, frequently communicate with other servers in other domains by establishing a new TCP connection (arrow 0). The recipient server establishes a port on which it listens for the connection, and with each connection the recipient learns the identity of the sender (its IP address, roughly equivalent to its SOP domain).

MashupOS follows the VOP, extending this facility to browser-side components by providing two additional communications paths: cross-domain browser-to-server communication and cross-domain browser-side communication.

1) Browser-to-server communication: Section II describes how the SOP protects legacy servers (such as those behind corporate firewalls) by confining browser-to-server communication to stay within the same SOP domain. We are not the first to observe [1], [6], [12] that cross-domain browser-to-server communication (arrow 2) can be safely allowed, so long as the proposed protocol labels the request with the domain that initiated it, and ensures that any participating server understands that it must verify the domain initiating the request. In particular, any VOP-governed protocol must fail with legacy servers; we adopt the technique proposed by JSONRequest [6], requiring servers to indicate their compliance by tagging their replies with a special MIME content type (application/jsonrequest). The JSONRequest proposal disallows the automatic inclusion of cookies with request transmission to avoid a variety of subtle vulnerabilities; our CommRequests obey the same restriction.

2) Browser-side communication: MashupOS also provides browser-side communication across domains (arrow 3). For example, a ServiceInstance from MashupOS may declare a port "inc", and register a handler function to receive browser-side messages on that port:

```javascript
function incrementFunc(req) {
  var src = req.domain;
  var i = parseInt(req.body);
  return i+1;
}
var srv = new CommServer();
srv.listenTo("inc", incrementFunc);
```

Another domain alice.com can address a browser-side message to Bob’s port using the new URL scheme local that specifies Bob’s SOP domain ((scheme, DNS host, TCP port) tuple) and port name:

```javascript
req = new CommRequest();
req.open("INVOKE",
  "local:http://bob.com/inc", false);
req.send(7);
y = parseInt(req.responseBody);
```

Local requests do not use HTTP, hence the special method INVOKE. Because the request is local, the implementation can forego marshaling objects into JSON or XML; instead, it need only validate that the sent object is data-only. As in JSONRequest [6], a data-only object is a raw data value, like an integer or string, or a dictionary or array of other data-only objects. The example false parameter specifies a synchronous request.

The port-based naming scheme works well for naming unique instances of services. If one service may be instantiated multiple times in a browser, however, then it is important to be able to address a particular instance by its relationship to the caller. For example, suppose both Alice’s page and Bob’s page include an instant-messaging gadget from im.com. Each parent page communicates with its own im.com ServiceInstance to set default parameters, or to negotiate Friv boundaries (Section V-C).

To facilitate communication addressed from parent to child or vice versa, MashupOS labels each ServiceInstance with a unique number (like a process ID).

```javascript
ServiceInstance.getId()
```

A ServiceInstance can register this identifier as a port name.

A ServiceInstance wishing to address its parent can do so by constructing the destination’s local: URL using these methods:

```javascript
url = "local:"
+ServiceInstance.parentDomain()
+ServiceInstance.parentId();
```

A ServiceInstance wishing to address its child can do so with these methods on the ServiceInstance element representing the child in the parent’s DOM:
The Web Applications working draft [31] proposes an alternative cross-domain browser-side communications mechanism. It provides only parent-to-child addressing, not global addressing between arbitrary browser-side components (like the local: scheme in MashupOS). It does not yet specify data-only communications (although we surmise that is the intent). It reveals the full URI (not just the domain) of the sending document, which may be inappropriate if the URI contains secret information such as session identification. It offers a unidirectional model, whereas CommRequest offers an asynchronous procedure call consistent with the XMLHttpRequest used in today’s deployed AJAX applications.

In summary, the new ServiceInstance tag unifies some existing browser features, some state of the art proposals, and some novel features to create a process-like abstraction for the browser environment. Besides providing a unified treatment of resources, the ServiceInstance introduces a consistent treatment of cross-domain communication, a flexible cross-domain display abstraction, intra-domain fault containment, and daemon processes.

VI. RESTRICTED SERVICES AND THEIR USAGE

In MashupOS, we enable service providers to offer restricted services (Section IV) that host third-party services or components and to enforce the restricted use of them. Allowing differentiation between restricted content and public content has significant security benefits as we will discuss in Section VII. In this section, we present how restricted services should be provided and used with restriction.

A. Restrictions for Restricted Services

Restricted services can contain any HTML or media content. However, they are not allowed to have direct access to any principals’ resources including their HTML DOM objects, cookies, nor to any principals’ remote data store at their backend Web server through XMLHttpRequest. Note that this is a one-way restriction that constrains restricted services from their integrators. The other direction of the restriction is at the discretion of integrators (which will be addressed in Subsection VI-C and Subsection VI-D).

Restricted services are allowed to communicate using CommRequest for both cross-domain browser-to-server communication and cross-domain browser-side communication. The origins of restricted services in such communications are marked as restricted, and the protocol requires participating Web servers to authorize the requester before providing service. Because the requester is anonymous, no participating server will provide any service that it would not otherwise provide publicly.

B. Hosting Restricted Services

We require a service provider to host restricted content, say “restricted.r”, differently from other public HTML content so that no browsers will render “restricted.r” as a public HTML page.

Restricted services can contain any HTML or media content. However, they are not allowed to have direct access to any principals’ resources including their HTML DOM objects, cookies, nor to any principals’ remote data store at their backend Web server through XMLHttpRequest. Note that this is a one-way restriction that constrains restricted services from their integrators. The other direction of the restriction is at the discretion of integrators (which will be addressed in Subsection VI-C and Subsection VI-D).

Restricted services are allowed to communicate using CommRequest for both cross-domain browser-to-server communication and cross-domain browser-side communication. The origins of restricted services in such communications are marked as restricted, and the protocol requires participating Web servers to authorize the requester before providing service. Because the requester is anonymous, no participating server will provide any service that it would not otherwise provide publicly.

We require a service provider to host restricted content, say “restricted.r”, differently from other public HTML content so that no browsers will render “restricted.r” as a public HTML page. Otherwise, “restricted.r” could be maliciously loaded into a browser window or frame (say, named “iframe”) without the constraints that are intended for restricted services (see above). The supposedly restricted service in “iframe” would have the same principal as the provider’s web site and access the provider’s resources. This violates the semantics of restricted services and can be exploited by attackers for phishing.

To prevent this, we employ the MIME protocol [10]. We require providers of restricted services to indicate their MIME content subtype to be prefixed with x-restricted+. For example, text/html content must be labeled text/x-restricted+.html.

C. Abstraction for Asymmetric Trust

When a service integrator consumes a restrictive service, it has at least an asymmetric trust with the service (Section IV). Another scenario of asymmetric trust is when an integrator uses a provider’s public library service, the integrator may want to access the library freely, but deny or control the library’s access to its own resources. In this section, we present our sandbox abstraction for asymmetric trust.

We introduce a new sandbox tag:

```
<Sandbox src='someContent'>
  <Fallback if sandbox tag not supported></Fallback>
</Sandbox>
```

The “src” file can either be a library service from a different domain or restricted content from any domains. However, we disallow a library service from the same domain as the integrator to be used here. This reason is that if the library were not trusted by its own domain, it should not be trusted by others, either.

The integrator should take caution to sandbox third-party libraries consistently — if a third-party library is sandboxed in one application, but not sandboxed in another application of the same domain, then the library can escape the sandbox when both applications are used. This scenario is similar to the issues of hosting restricted services discussed in Section VI-B.

With the sandbox abstraction, although the sandboxed content cannot reach out of the sandbox, the enclosing page of the sandbox can access everything inside the sandbox by reference. The access includes reading or writing script global objects, invoking script functions, and modifying or creating DOM elements inside the sandbox. The enclosing page cannot put its own object references (or any other references that do not belong to the sandbox) into the sandbox. This is to prevent code from within the sandbox to follow those references out of the sandbox. For example, the enclosing page is not allowed to

Note that file extensions do not dictate the content type to be rendered; MIME types plus file content often determines how the files are rendered in today’s browsers [13], [15], [26].
pass its own display elements into the sandbox. If an integrator wants to integrate a third-party library together with some of its own content such as display elements that may be needed by the library, the integrator should create its own restricted content that include both the library and the display elements, and then sandbox that restricted service.

Sandboxes can be nested. A sandbox’s ancestors can access everything inside the sandbox; and the sandbox still cannot access anything outside of it. It follows that sandbox siblings cannot access one another. Any DOM elements can be enclosed inside a sandbox including ServiceInstance. However, a ServiceInstance declared inside a sandbox does not give the ServiceInstance any additional constraints. No matter where a ServiceInstance is located in a DOM tree, it always represents a service instance of a principal, which shares with other instances of the same principal the persistent state of the principal and the remote data on its web server. Therefore, the sandbox cannot access any resources that belong to its child ServiceInstances.

Sandboxes are particularly useful for creating robust client mashups out of third-party library services. For each third-party library service that a service integrator uses, the integrator can create a restricted service enclosing the library along with its needed display DOM elements such as div and put the restricted service into a sandbox. The integrator can then access and mash up content across its sandboxes as it wishes without worrying about any of the library services maliciously or recklessly tampering with the integrator’s content or other resources. Most of today’s client mashups use third-party scripts for service components. In Section IX-A, we demonstrate how easy it is to implement a more secure client-side mashup with Sandbox.

1) Traditional Sandboxing for Desktop Applications: Our sandbox abstraction is reminiscent of the original sandboxing mechanism introduced by Wahbe et al. [30]. Wahbe et al. address the scenario where desktop applications contain (untrusted) third-party libraries or modules. They use software-based fault isolation (SFI) to sandbox or isolate faults of an untrusted module in the same address space as an application’s process: They load the untrusted module into a fault domain which is a segregated and contiguous region of memory within an address space and then use binary rewriting to modify the object code of the untrusted module to prevent it from writing or jumping to an address outside its fault domain. Program modules isolated in separate fault domains cannot modify each other’s data or execute each other’s code except through an explicit cross-fault-domain RPC interface [30].

The primary motivation for the original sandboxing work was to avoid the high overhead of context switches if the untrusted code were placed in a separate process, rather than to have an abstraction for realizing asymmetric trust. In fact, people have not paid much attention to or leveraged the property of asymmetric trust and access in the original sandboxing. In MashupOS, we adapt sandboxing to the browser environment to particularly leverage this asymmetry to match a common trust scenario between providers and integrators and provide both security and ease in creating client mashups.

2) Risks: With the sandbox abstraction, the code from within a sandbox can never follow references to outside the sandbox. However, the data references from within a sandbox can be used by the outside of the sandbox. Since these data references are also managed by the untrusted library service, Web programmers of a service integrator should take caution and check the validity of these data before use. If the programmers find this dependency risky, or are concerned that later maintainers of the code will err, then this manifests a controlled trust scenario rather than an asymmetric trust scenario. The reason is that both the enclosing page accessing the library and the library accessing the enclosing page need to be tightly controlled; and access control for the former is additionally demanded. The next section addresses this scenario. Essentially, the programmers force themselves to use explicit data communication for accessing restricted content, and hopefully this will help them be more disciplined in checking the data’s validity.

D. Abstraction for Using Restricted Services with Controlled Trust

We use our ServiceInstance again for controlled trust with restricted services where an integrator also wants to control its own access to restricted services. When the MIME type of a ServiceInstance’s content indicates restricted content, the ServiceInstance automatically disallows its content from accessing the resources of the ServiceInstance’s hosting domain including XML-HTTPRequest and cookie access, in addition to HTML DOM isolation that ServiceInstance already provides.

This restricted mode of ServiceInstance is the same as the <Module> tag [7] except that unlike the <Module> proposal, we allow the ServiceInstance to communicate using both forms of CommRequest.

VII. COMBATTING CROSS SITE SCRIPTING ATTACKS

As of 2006, more than 21% of vulnerabilities reported to CVE are Cross Site Scripting (XSS) vulnerabilities, ranking number one and surpassing long-running champion, buffer overflows, for two years in a row [4]. In this section, we show how we use our abstractions ServiceInstance and Sandbox to combat XSS attacks in a fundamental way while retaining or even enhancing the richness of page presentation.

A. Background on XSS Vulnerabilities

XSS is a type of vulnerability found in web applications. In XSS, an attacker often exploits the case where a web application injects user input into a dynamically generated pages, without first filtering the input [20]. The injected content may be either persistent or non-persistent. As an example of a persistent injection attack, an attacker uploads a maliciously-crafted profile to a social networking web site. The site injects the content into pages shown to others who view the profile. An injected script runs with the social networking site as its domain, enabling the script to make requests back to
the site on behalf of the user. The notorious Samy [29] worm that plagued myspace.com exploited persistent injection; it infected over one million myspace.com user profiles within the first twenty hours of its release.

A malicious input may also be non-persistent, simply reflected through a web server. For example, suppose a search site replies to a query $x$ with a page that says “No results found for $x$.” An attacker can trick a user into visiting a URL which contains a malicious script within the query $x$ to the search site. The script in the reflected page from the search site will run with the search site’s privilege.

**B. Existing Defense**

The root causes of the XSS are unsanitized user input and unexpected script execution. Many existing mechanisms tackle the first cause by sanitizing user input. For applications that take text-only user input, the sanitization is as simple as enforcing the user input to be text, escaping special HTML tag symbols like < into their text form like “&lt;”. However, many web applications, such as social networking Web sites like myspace.com, demand rich user input in the form of HTML. Because no existing browser abstractions constrain the reach of an included script, these web sites typically have the policy of denying scripts in the user uploaded HTML pages. Consequently, user input sanitization involves script detection and removal. However, this turns out to be non-trivial: Because browsers speak such a rich, evolving language and many browser implementations exist and vary, there are many ways of injecting a script [27]. In many occasions already, creative attackers have found new ways of injecting a script. The Samy worm [29] was notorious for discovering several holes in myspace.com’s filtering mechanism.

The difficulty of exhaustive input filtering led researchers to tackle the second root cause, preventing unexpected script execution. Jim et al. [20] proposed BEEP to white-list known good scripts and adding a “noexecute” attribute to $<\text{div}>$ elements to disallow any script execution within that element. This work makes a good step towards the servers’ goal of denying scripts. One drawback of this approach is its insecure fallback mechanism when BEEP-capable pages run in legacy browsers: The “noexecute” attribute would be ignored by legacy browsers, allowing scripts in the $<\text{div}>$ element to execute.

**C. Using MashupOS for Defense**

The reason behind Web servers’ policy to disallow scripts is that existing browsers provide no way to restrict a script’s behavior once it is included. The best known approach is to use a cross-domain iframe to isolate the user-supplied scripts. This approach is undesirable for three reasons:

- It requires the server to serve the scripts from a second domain (to associate the scripts with a distinct domain),
- The iframe provides an inflexible display layout, and
- The user-supplied content cannot interact, even in a constrained way, with its containing page.

MashupOS’s Sandbox and ServiceInstance solve all of the problems with iframe and can serve as a fundamental defense against XSS while allowing third-party script-containing rich content. The choice of the abstraction used depends on whether the trust model is asymmetric trust or controlled trust (Section VI-C and Section VI-D).

For persistent, user-supplied HTML content, a Web server can serve it as restricted content (Section VI-B). For non-persistent user input in a reflected server page, the reflected page can use a Sandbox or ServiceInstance to contain the user input with the “src” attribute being either a dynamic page proxied by the server:

```html
<Sandbox src='userInput.asp?... escaped input ...'/></Sandbox>
```

or a “data” URI [24] with encoded content:

```html
<Sandbox src='data:text/x-restricted+html, ... escaped user input ...'/>
```

A ServiceInstance enables flexible layout by connecting the restricted content’s display to the parent container with a Friv. In the case of sandbox, the restricted content’s display DOM is directly accessible by the parent. A ServiceInstance can communicate with its parent’s client or server components using the CommRequest primitive. A sandbox can do the same, plus the parent can communicate with the child by directly accessing the child JavaScript objects.

**VIII. IMPLEMENTATION**

We have built a MashupOS prototype system in which we have realized almost all proposed abstractions and their properties except the feature of decoupling display from ServiceInstance. Our implementation requires one Friv per ServiceInstance (which may contain child Frivs or frames). Our prototype is based on Internet Explorer 7 (IE) and runs...
on both Windows XP SP2 and Windows 2003 SP1, but our methodology and techniques can also be applied to other browsers.

Instead of modifying IE’s source code directly, we leverage browser extensions and public interfaces exported by IE. Figure 3 shows the MashupOS extensions. Our system consists of two extensions to the IE architecture [14].

The first extension is a script engine proxy that we built from scratch using public interfaces exported by IE. As in all browsers, IE consists of an HTML/CSS rendering and layout engine and various script engines including a JavaScript engine and a VBScript engine. When a script element is encountered during HTML rendering, the script element is handed to a corresponding script engine for parsing and execution. Script execution may manipulate HTML DOM objects. For this purpose, the script engine asks the rendering page for references to needed DOM objects. We introduce the mechanism of script engine proxy (SEP), which interposes between the rendering engine and the script engines, and mediates and customizes DOM object interactions. To the rendering engine of a browser, a SEP serves as a script engine and exports the interface of a script engine; to the original script engine of the browser, the SEP serves as a rendering engine and exports the DOM interface of the rendering engine. We use object wrappers for the purpose of interposition. When a script engine asks for a DOM object from the rendering engine, a SEP intercepts the request, retrieves the corresponding DOM object, associates the DOM object with its wrapper object inside the SEP, and then passes the wrapper object back to the original script engine. From that point on, any invocation of the wrapper object methods from the original script engine goes through the SEP. In IE, a SEP takes the form of a COM [5] object and is registered in Windows Registry associated with a scripting language (such as JScript) to serve as IE’s script engine for that language. We have focused on the JavaScript language which is dominant in today’s web applications. Our techniques can be readily applied to other types of languages.

Our MashupOS Script Engine Proxy (MSEP) takes the crucial role of implementing our various protection abstractions. Our general strategy here is to use the existing isolation mechanism, namely cross-domain frames, as our building block. A ServiceInstance element is implemented using a cross-domain frame with at least one script in it (so that we can trigger MSEP’s interposition on the frame). Even ServiceInstances from the same domain are segregated as cross-domain frames, though they are allowed to access the same set of cookies. A sandbox element is also implemented using a frame. Accessing from the outside to the inside of the frame mimics that of same-domain frames, while accessing from the inside to the outside of the frame mimics that of cross-domain frames. We further mediate each object access from within a sandbox to ensure that the object belongs to the sandbox and not a reference from the outside of the sandbox. Our customized access control of cross-domain frames for realizing our abstractions is realized via object wrappers as described above.

The second extension, MashupOS MIME filter, is an asynchronous pluggable protocol handler [25] at the software layer of URLMon.dll where various content (MIME) types are handled. Our MIME filter takes an input HTML stream and outputs a MashupOS-transformed HTML stream to the next software layer in IE. We use our MIME filter to translate new tags into existing tags, such as iframe and script; and we use special JavaScript comments inside an empty script element to indicate the original tags and attributes to MSEP. For example,

```
<iframe src='restricted.rhtml'
    name='s1'>
</sandbox>
```

is translated by our MIME filter to

```
<iframe src='restricted.rhtml'
    name='s1'>
</iframe>
```

The comments inside the script element informs MSEP that the iframe with name “untrustedSandbox” should be treated as a sandbox. Similar translation happens to ServiceInstance.

Our CommRequest-based communication primitives are implemented by providing two runtime objects CommServer and CommRequest, with the communication methods described in Section V-D. For access control on XMLHttpRequest, particularly for restricted mode ServiceInstances and Sandboxes, we again use the object wrapper mechanism above for interception.

friv is implemented using iframe as well. We used CommRequest to carry out the automatic negotiation on the frame width and height between a friv and its parent.

We find that script engine proxies can serve as a great platform for experimenting with new browser features. The fact that we are able to implement all our abstractions on this platform along with the MIME filter indicates that they should also be easy to add to the existing browsers.

IX. Evaluation

In this section, we first demonstrate the ease of programming robust web services with MashupOS protection abstractions by showcasing an example application in Section IX-A. Then, we report the performance measurement of our prototype system in Section IX-B.

A. Showcase Application

We have implemented a photo location web service, called PhotoLoc, as a showcase application. PhotoLoc mashes up Google’s map service [11] and Flickr’s geo-tagged photo gallery service [9] so that a user can map out the locations of photographs taken. Flickr provides API libraries in languages
like Java or C# to interact with the Flickr Web server across the network, for example to retrieve geo-tagged photographs. To facilitate easy, browser-side cross-domain communication, we created an access-controlled, Flickr-based service that can be accessed using our Friv for isolation and communication. The trust relationship with any access controlled service is controlled trust. Therefore, PhotoLoc and Flickr must communicate with each other through CommRequest (Section V-D). Google’s map service is a public library utility service (Section IV). PhotoLoc chooses an asymmetric trust relationship with Google’s map library, that is it trusts itself to use the library, but does not trust the library to access PhotoLoc’s resources. PhotoLoc puts Google’s map library along with the Div display element that the library needs into “g.uhtml” and serves “g.uhtml” as restricted content. PhotoLoc’s main service page (index.htm) uses Sandbox to contain “g.uhtml”. PhotoLoc can access everything inside the sandbox, but Google’s map library cannot reach out of the sandbox. Figure 4 shows our implementation.

**B. Performance**

Now we present the performance measurement of our MashupOS prototype. We first present our micro-benchmark results that measure the overhead of our MashupOS-enabled Script Engine Proxy (MSEP). Then, we present our macro-benchmark results on the impact of MSEP on page loading time. We conducted our measurements on a Pentium 4 PC of 1.7 GHz CPU and 1.5 GB of RAM, which runs Windows XP SP2 and Internet Explorer (IE) 7.

1) **Microbenchmark on MSEP Overhead:** We found a JavaScript and DHTML script performance benchmark called BenchJS [2]. The benchmark contains the following 7 JavaScript and DHTML tests.

1) Counting: count to 10000 and display a progress bar.
2) Open pops: open 8 pop ups and close them.
3) Replace images: replace 300 tiny images as fast as possible. It repeats this procedure 10 times.
4) Text manipulation: manipulate long text with different ways.
5) Set tables: create 2000 table-cells and calculates a random background color for each table-cell.
6) Put layers into place: create a phrase out of 50 different layers that are pulled together.
7) Calculate x-mas: calculate the days of the week for the next 10000 x-mas (not counting display time).

We run the 7 tests on both a MSEP-equipped IE 7 and IE 7 alone. We run each test for 4 times, and report the average
latex in the top half of the Table IX-B.1. As we can see, for both the computational tests (Test 1, 4, and 7) that involve pure JavaScript objects and the tests with moderate interactions with DOM objects (Test 2 and 5), we observe negligible overhead. This is because MSEP’s involvement in them is little. Test 3 and 6 involve heavy interactions with DOM objects and incur 82% and 33% overhead due to our DOM object interposition and manipulation.

To further understand MSEP’s overhead targeting its interposition functions, we designed our own set of microbenchmarks as follows:

1) Pure JavaScript object: All 14 properties and methods of “Number” object.
2) DOM object: 92 properties and methods of “window”, “document” and “clientInformation” object.
3) Complex DOM operation: 4 JavaScript statements including DOM traversal, element creation, style sheet update, inline script library loading, and event firing.
4) Communication: Cross-frame (same domain) function invocation when without MSEP; Cross-frame communication via CommRequest when with MSEP.

We measured the duration for 10,000 runs of each individual script statement with and without MSEP. We report the average time in the bottom half of Table IX-B.1. The results are consistent with that of BenchJS. MSEP incurs noticeable overhead for DOM manipulations, but negligible overhead for pure JavaScript object manipulations. Also, MashupOS’s CommRequest is even more efficient than cross frame function calls.

2) **Macrobenchmark Results:** Our Macrobenchmark measurement evaluates the impact of MSEP on the overall page loading time. We picked the top 500 pages from the top click-through search results of a major search engine from 2005. In our measurement, we disabled the browser cache, measured the page loading with and without MSEP. We implemented a Browser Helper Object [3] to capture the time before a document navigation and after the document loading. From our numbers, we do not observe any MSEP’s impact on the page loading time. Among the 500 pages that we measured, 30 of them cannot be loaded or rendered correctly in IE 7, and MSEP caused just one more faulty page than that. This gives us much confidence on the completeness and robustness of our prototype.

### X. RELATED WORK

#### A. Isolation abstractions using existing browsers

Recently, a new wave of “web operating systems” [8] (e.g., YouOS [32]) have emerged. These sites present a traditional desktop user interface, complete with a window manager. The applications run natively in JavaScript. All are hosted on the same domain as the web desktop, and thus have unlimited access to one another. This lack of isolation, comparable to the 1995 PC desktop, requires the user to completely trust every application.

A recent system called Subspace [19] provides a cross-domain communication mechanism that is designed to run on current browsers without any additional plug-ins or client-side changes. Subspace splits sites into subdomains for each principal, which can in turn be used to draw in scripts from other domains. A construction based on the browser’s document.domain setting is used to pass data safely between subdomains. However, Subspace requires significant work on the part of the web developer to use correctly, particularly for complex mashups with untrusted code from many different sources. We believe browsers should provide built-in isolation and communication primitives.

#### B. Verifiable-Origin Policy Communication Proposals

MashupOS’s communication mechanism is inspired by several earlier proposals for cross-domain communication governed by a verifiable-origin policy.

The JSONRequest [6] is a proposed VOP communication mechanism for network requests that transmit data in JSON format (a data-only subset of JavaScript). Using the VOP instead of the SOP is safe because cookies are not sent, because the request includes a header indicating the source of the request, and because the server’s reply must indicate the server is aware of the protocol and hence its security implications.

A new HTML module tag has been proposed to partition a page into a collection of modules [7]. A module groups DOM elements and scripts into an isolated environment; VOP communications can be used to transmit data in JSON format between the inner module and the outer module. Thus, modules are similar to restricted ServiceInstances.

Cross-document messages [12] are a proposed browser standard that would allow frames to send VOP messages to each other. They are implemented in the Opera browser, although currently only strings can be sent.
The Flash browser plugin offers a less fine-grained form of VOP communication using cross-domain policy files [1]. A service provider can place a crossdomain.xml file on the server, and Flash uses the contents of this file to determine which service integrator domains can access files on the provider’s server. The security of this technique relies on the assumption that the attacker cannot place a malicious crossdomain.xml file on a victim’s server.

XI. CONCLUDING REMARKS

The advent of AJAX and client mashups have turned Web browsers into a multi-principal operating environment. However, browser support for Web programmers has lagged behind and remained in a single-principal world. The MashupOS project is building a multi-principal operating system for Web browsers. This paper focuses on the most imminent needs of today’s browsers: abstractions for protection and communication.

Our analysis shows that three types of trust exist between Web service providers and integrators: full trust, controlled trust, and asymmetric trust. Existing browser abstractions support only a binary trust model: no trust (which is just one configuration of controlled trust) through the use of cross-domain frames for complete isolation denying any communications, or full trust by including third party scripts as libraries allowing full resource access.

In MashupOS, we have proposed ServiceInstance and Sandbox as new isolation abstractions realizing controlled trust and asymmetric trust, and CommitRequest as a VOP-based communication abstraction unifying existing cross-domain communication proposals.

We have also proposed restricted services as a fundamental addition to today’s Web service provisioning. Our abstractions enable Web service providers to offer restricted services and to enforce their restrictions. Such support for restricted services can fundamentally combat Cross Site Scripting attacks (a prominent threat in today’s Web) while allowing the richest possible third party content.

Our abstractions are backward compatible allowing Web programmers to supply alternative content for browsers that don’t support our abstractions. We have carefully designed the MashupOS abstractions to avoid unintended interactions between new services that use our abstractions and legacy ones. These enable Web programmers to adopt our abstractions with ease and comfort.

Our MashupOS prototype realizes almost all our proposed abstractions and their properties. Our evaluation showcases an easy-to-build and robust client mashup. Measurement of our prototype shows negligible overhead. The implementation and evaluation demonstrate the ease of adding these abstractions to existing browsers.

REFERENCES